

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

Java Persistence with Hibernate

Second Edition

Hibernate

实战

(第2版)



[德] Christian Bauer

[澳] Gavin King

[美] Gary Gregory

蒲成

著
译

MANNING



清华大学出版社

Hibernate 实战

(第2版)

[德] Christian Bauer

[澳] Gavin King 著

[美] Gary Gregory

蒲成 译

清华大学出版社

北京

Christian Bauer, Gavin King, Gary Gregory
Java Persistence with Hibernate, Second Edition
EISBN: 978-1-61729-045-9

Original English language edition published by Manning Publications, 178 South Hill Drive, Westampton, NJ 08060 USA. Copyright©2016 by Manning Publications. Simplified Chinese-language edition copyright © 2016 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Manning 出版公司授权清华大学出版社独家出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。
版权所有, 侵权必究。

北京市版权局著作权合同登记号 图字: 01-2016-6623

本书封面贴有清华大学出版社防伪标签, 无标签者不得销售。
版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

Hibernate 实战: 第2版 / (德)克里斯蒂安·鲍尔(Christian Bauer) 等著; 蒲成 译. —北京: 清华大学出版社, 2016

书名原文: Java Persistence with Hibernate, Second Edition

ISBN 978-7-302-44808-2

I. ①H… II. ①克… ②蒲… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2016)第 192650 号

责任编辑: 王 军 于 平

装帧设计: 思创景点

责任校对: 曹 阳

责任印制: 刘海龙

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 邮 编: 100084

社 总 机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者: 清华大学印刷厂

经 销: 全国新华书店

开 本: 185mm×260mm 印 张: 33.25 字 数: 809 千字

版 次: 2016 年 9 月第 1 版 印 次: 2016 年 9 月第 1 次印刷

印 数: 1~3500

定 价: 88.00 元

产品编号: 068422-01

第 1 版的赞誉

Hibernate 以及企业计算领域中对象/关系映射的权威指南。

——来自 Linda DeMichiel 所写的第 1 版序言

Sun Microsystems

本书是终极的解决方案。如果你打算在你的应用程序中使用 Hibernate，那么别无选择，赶紧去书店购买本书吧。

——Meera Subbarao

JavaLobby

本书是最全面、权威的 Hibernate 指南，并且是 OR 映射 Java 持久化方面的权威指南、辅导教程以及参考资料。

——Will Wagers

C#online.net

Hibernate 方面的权威资料。非常适合所有的开发人员。

——Patrick Peak, CTO

BrowserMedia, *Hibernate Quickly* 的作者

我竭诚推荐这本书！

——Stuart Caborn

ThoughtWorks

非常棒的主题，非常棒的内容——非常需要这本书！

——Ryan Daigle, RTP Region

ALTERthought

这是市面上内容最全面的 Hibernate 书籍。它涵盖了方方面面的内容。

——Liz Hills

Amazon reviewer

译者序

现代的企业级应用从诞生之日起就与数据密不可分，而数据库系统(尤其是关系型数据库系统)更是软件开发领域无法绕过的一环。如何高效建立数据模型以及在应用程序中使用对象编程思想操纵数据，一直是广大开发人员孜孜不倦寻求完美解决方案的目标，而 Hibernate 的诞生正是为了满足这样的需求。

作为一个开源的对象关系映射框架，Hibernate 对 JDBC 进行了极为轻量级的对象封装，这样一来，Java 开发人员就可以随心所欲地应用对象编程思想操作关系型数据库。Hibernate 可以应用在任何使用 JDBC 的场景中，既可以在 Java 的客户端程序中使用，也可以在 Servlet/JSP 的 Web 应用中使用。最具有革命性意义的是，Hibernate 可以在应用 EJB 的 J2EE 架构中取代 CMP，承担起数据持久化存储引擎的重任。当然，这也是当年 Gavin King 发起 Hibernate 项目的初衷。相比 Hibernate 来说，EJB 显得太过重量级了。并且随着新技术和新框架的发展，最新的 Spring + Struts + Hibernate 这一组合在很大程度上已经可以取代 EJB 了。

多年来的实践证明，Hibernate 是专门针对 Java 语言所创建的最优秀的持久化存储引擎之一。越来越多的开发人员都从应用 Hibernate 中受益，并且越来越多的持久化存储引擎也都采用了 Hibernate 的先进思想和理念。而本书正是致力于将 Hibernate 这一技术思路推而广之的指南和参考资料，它能让你全面理解如何在自己的项目中应用 Hibernate。

本书使用了大量贴合实际的示例，结合作者的亲身实践和多年的丰富经验来介绍 Hibernate 的概念与应用原则。重点在于让读者理解其价值和实现的机制及原理，而不是空谈枯燥乏味的理论知识点。

本书详细阐述了 Hibernate 的各方面实践和各种可用工具，采用示例方式为读者呈现从头至尾完整构建一个以 ORM 为核心的应用程序的过程。诚然，Hibernate 也有其自身的局限和不足，但随着软件开发领域的推陈出新，其开发团队也在不知疲倦地对 Hibernate 进行更新和调整，使其能跟上时代发展的步伐。作为一本 Hibernate 的权威书籍，本书能让你快速、专业地掌握以 Hibernate 为基础构建 ORM 的技术。

在此要特别感谢清华大学出版社的编辑，在本书的翻译过程中他们提供了很多帮助，没有他们的热情付出，本书将难以付梓。本书全部章节由蒲成翻译，参与本次翻译的还有何东武、李鹏、李文强、林超、刘洋洋、茆永锋、潘丽臣、王滨、申成龙、王佳、潘勇、负书谦、杨达辉、赵永兰、郑斌、杨晔。

译者

第 1 版序言

关系数据库无疑是现代企业的核心。当包括 Java 在内的现代编程语言提供应用级业务实体的面向对象的直观视图时，这些实体背后的企业数据实质上就是关系型的。此外，关系模型的主要优势在于——相对较早的导航模型和之后的 OODB 模型而言——根据设计，它在本质上对于所服务的程式操作以及数据的应用级视图是不可知的。在桥接关系与面向对象技术方面，我们已经做过太多尝试，但这两者之间的差距是当今企业计算的一个无可辩驳的事实。Hibernate 面临通过其对象/关系映射(ORM)方法来充当关系数据和 Java 对象之间桥梁的挑战。Hibernate 以一种非常实用、直接且现实的方式迎接了这一挑战。

正如 Christian Bauer 和 Gavin King 在本书中所揭示的那样，要在所有最简单的企业环境中有效使用 ORM 技术，需要理解和配置关系数据与对象之间中介的执行方式。这要求开发人员认识到并理解应用程序及其数据需求，以及 SQL 查询语言、关系型存储结构与关系技术提供的优化潜力。Hibernate 不仅为满足这些需求提供了功能完整的解决方案，它还是一个灵活且可配置的架构。Hibernate 的开发人员在设计它的时候考虑到了模块性、可插入性、可扩展性以及用户自定义。因此，从其第一个版本发布之后的数年间，Hibernate 迅速成为企业开发人员的一种主导 ORM 技术。

本书提供 Hibernate 的综合概述，涵盖下面这些内容：如何将 Hibernate 的类型映射功能用于关联和继承建模；如何使用 Hibernate 查询语言有效检索对象；如何配置 Hibernate 以便同时用于托管和非托管环境中。此外，在本书中作者通篇为 ORM 底层问题以及 Hibernate 背后的设计选择提供其见解。这些见解让读者能够深刻理解 ORM 作为企业级技术的有效使用。《Hibernate 实战(第 2 版)》是在如今的企业计算中使用 Hibernate 以及对象/关系映射的权威指南。

Linda Demichiel

首席架构师，Enterprise JavaBeans
Sun Microsystems

序 言

这是我们出版的关于 Hibernate 的第 3 本书, Hibernate 是一个将近 15 岁的开源项目。最近一次调查显示, Hibernate 是很多 Java 开发人员每天都使用的排名前五的工具。这表明, SQL 数据库仍旧是可靠数据存储和管理的首选技术, 对于 Java 企业软件开发领域来说尤为如此。

现在已经发布了 Hibernate 的第 5 个主要版本, Hibernate 实现的 Java 持久化 API (Java Persistence API, JPA) 规范的第 2 个主要版本也已经发布了。Hibernate 的核心, 现在称为对象/关系映射 (Object/Relational Mapping, ORM) 的部分, 已经成熟应用了很长时间, 并且这些年来已经做出许多细小的改进。其他的相关项目, 例如 Hibernate 搜索、Hibernate Bean 验证以及最新的 Hibernate 对象/网格映射 (Object/Grid Mapping, OGM), 都是新交付以及创新的解决方案, 它们使得 Hibernate 成为用于各种数据管理任务的完整工具套件。

在编写本书的上一版时, Hibernate 正在经历一些重大变化: 稳步壮大并由开源社区以及 Java 开发人员的日常需求所驱动, Hibernate 必须变得更为规范并实现 JPA 规范的第一个版本。因此上一版是一本很厚的书, 因为许多示例都必须同时以旧的形式和新的标准化形式来给出。

如今此差距几乎完全消失了, 我们现在可以首要借助 Java 持久化的标准 API 和架构。当然还有许多很好的 Hibernate 功能, 我们将在这一版中进行探讨。尽管与上一版相比, 本书的页数减少了, 但我们将这一空间留给了为数众多的新示例。我们还将介绍 JPA 如何适用于较大的 Java EE 场景, 以及你的应用程序架构如何才能集成 Bean 验证、EJB、CDI 以及 JSF。

让本书成为你的首个 Hibernate 项目的指南。我们希望它会替代上一版作为你书桌上的 Hibernate 参考文档。

在本书的编写过程中, 我们得到了许多人的帮助。在本书的翻译过程中他们提供了很多帮助。没有他们的热情付出, 本书将难以付梓。本书全部章节由周成翻译, 参与本次翻译的还有何东武、李毅、李文强、林磊、刘伟、高永强、陈超、王佳、潘亮、周书敏、靳建辉、赵永兰、郑斌、杨峰。

在此要特别感谢清华大学出版社的编辑, 在本书的翻译过程中他们提供了很多帮助。没有他们的热情付出, 本书将难以付梓。本书全部章节由周成翻译, 参与本次翻译的还有何东武、李毅、李文强、林磊、刘伟、高永强、陈超、王佳、潘亮、周书敏、靳建辉、赵永兰、郑斌、杨峰。

致 谢

如果没有许多人提供帮助，我们无法完成本书的撰写。作为本书的技术评审，Palak Mathur 和 Christian Alfano 表现得很出色；感谢他们花费了大量时间来编辑我们破碎的代码示例。

我们还要感谢审稿人，他们在开发阶段投入了很多时间，并且提供了珍贵的反馈：Chris Bakar、Gaurav Bhardwaj、Jacob Bosma、José Diaz、Marco Gambini、Sergio Fernandez Gonzalez、Jerry Goodnough、John Griffin、Stephan Heffner、Chad Johnston、Christophe Martini、Robby O' Connor、Anthony Patricio、Denis Wang。

Manning 出版社的出版人 Marjan Bace 再一次聚集了 Manning 的伟大制作团队：Christina Taylor 编辑了我们粗糙的原稿，并且将它变成了一本真正的书。Tiffany Taylor 找出了我们所有的拼写错误，并且让这本书具有了可读性。Dottie Marsico 负责本书的排版，并且让这本书有了极佳的观感。Mary Piergies 协调和组织了本书的制作过程。我们衷心感谢所有与我们一起工作的人。

最后，要特别感谢 Linda DeMichiel 为第一版编写了前言。

Gary Gregory

我希望感谢我的父母，他们开启了我的人生旅程，为我提供了受到良好教育的机会，并且让我能够自由选择我的人生。我将永远感激我的妻子 Lori 和我的儿子 Alexander，他们给了我时间来从事另一个项目，比如编写我的第三本书。

一路走来，我的学习和工作历程都有真正无与伦比的伙伴陪同，比如 George Bosworth、Lee Breisacher、Christopher Hansen、Deborah Lewis，以及其他许多人。值得一提的是，我的岳父 Buddy Martin 通过很棒的对话和来自于他数十年编写体育报道的经验为我提供了智慧和见解。最后，我要感谢我的合著者 Christian Bauer，他分享了自己的知识；还要感谢 Manning 的所有人，他们提供了支持、专业能力以及很好的反馈。

前 言

本书既是关于 Hibernate 和 Java 持久化的一本指南，也是一本参考资料。如果你才开始接触 Hibernate，我们建议你从本书第 1 章开始阅读，并且使用第 2 章的“Hello World”教程开始编码。如果你已经使用过较早的 Hibernate 版本，那么应该快速阅读前两章以便得到一个概览，然后跳到第 3 章的中间部分。

路线图

本书分为五个主要部分。

在第 I 部分“ORM 入门”中，我们将探讨对象/关系映射背后的基础。我们将演练亲身实践的指南，以便让你开始处理自己的首个 Hibernate 项目。我们将介绍用于域模型的 Java 应用程序设计，以及用于创建对象/关系映射元数据的选项。

第 II 部分“映射策略”专注于 Java 类及其属性，以及它们如何映射到 SQL 表和列。我们将探究 Hibernate 和 Java 持久化中的所有基本和高级映射选项。我们将介绍如何处理继承、集合以及复杂类关联。最后，我们要探讨遗留数据库模式的集成以及特别难处理的一些映射策略。

第 III 部分“事务性数据处理”完全与使用 Hibernate 和 Java 持久化加载与存储数据有关。我们将介绍编程接口、如何编写事务性应用程序，以及 Hibernate 如何才能最有效地从数据库加载数据。

第 IV 部分“编写查询”将介绍数据查询功能，并详尽讲解查询语言和 API。在这一部分中，并非所有章节都是以指南的风格来编写的；我们期望你在构建一个应用程序以及为特定查询问题查找解决方案时经常浏览本书的这一部分。

第 V 部分“构建应用程序”将探讨分层和有会话意识的 Java 数据库应用程序的设计与实现。我们要探讨用于 Hibernate 的最常见设计模式，比如数据访问对象(Data Access Object, DAO)。你会看到如何才能轻松测试自己的 Hibernate 应用程序，并且学习在 Web 和客户端/服务器应用程序中使用对象/关系映射软件时，通常有哪些相关的最佳实践。

读者对象

本书的读者应该具备基本的面向对象软件开发知识，并且应该在实践中已经应用过这一知识。为了理解应用程序示例，你应该熟悉 Java 编程语言以及统一建模语言。

我们的主要目标读者由使用基于 SQL 数据库系统的 Java 开发人员构成。我们将介绍如何通过使用 ORM 大幅提高你的生产效率。如果你是一位数据库开发人员，那么本书可

将你引入面向对象软件开发。

如果你是一位数据库管理员，那么你有兴趣了解 ORM 如何影响性能，以及如何才能调整 SQL 数据库管理系统和持久化层的性能来实现性能目标。由于数据访问是大多数 Java 应用程序中的瓶颈，因此本书密切关注性能问题。许多 DBA 都对于将性能委托给工具生成的 SQL 代码这一点感到紧张，这是可以理解的；我们力图缓解那些不安，并且突出介绍应用程序不应该使用工具托管的数据访问的用例。你可能会欣慰地发现，我们并没有宣称 ORM 是每一个问题的最佳解决方案。

代码规范

本书提供了丰富示例，其中包括所有的 Hibernate 应用程序构件：Java 代码、Hibernate 配置文件以及 XML 映射元数据文件。代码清单或文本中的源代码都使用了固定宽度的字体格式，以便将其与普通文本区分开来。此外，文本中的 Java 方法名称、组件参数、对象属性以及 XML 元素和属性也使用了固定宽度的字体格式。

Java、HTML 和 XML 都会很冗长。在许多用例中，原始的源代码已经被重新格式化；我们在本书中已经添加换行符和重新处理的行首缩进来调节可用的页面空间。在极少的用例中，即便如此也是不够的，因而清单包含了续行符标记(↵)。此外，当文本中描述了代码时，源代码中的注释通常就会从清单中移除。有些源代码清单伴随着代码注解，以突出重要的概念。

源代码下载

Hibernate 是基于宽松 GNU 公共许可来发布的一个开源项目。Hibernate 网站 www.hibernate.org 上提供了以源代码或二进制格式下载 Hibernate 包的指示。<http://jpwh.org/> 上提供了本书中所有示例的源代码。你也可以从本书封底的二维码下载本书中的示例代码。

作者在线

购买《Hibernate 实战(第 2 版)》就可以免费访问由 Manning 出版社运营的私有网络论坛，在该论坛中你可以发表与本书有关的评论，提出技术问题，并且接受来自作者及其他用户的帮助。要访问该论坛并订阅它，请将你的网络浏览器导航到 www.manning.com/books/java-persistence-with-hibernate-second-edition。这个页面提供了与注册之后如何访问该论坛、可以得到哪些帮助，以及该论坛执行的规则有关的信息。

Manning 向我们的读者承诺，将提供读者之间以及读者与作者之间可以进行有意义对话的场所。就作者而言，无法确保某种程度的参与度，他们对于该论坛的贡献仍然是自愿(且无偿)的。我们建议你尝试向作者提出具有一些挑战性的问题，以便他们对你的问题深感兴趣！

只要本书还在印刷，那么作者在线论坛以及之前探讨的记录就可以从出版者的网站上访问到。

作者简介

Christian Bauer 是 Hibernate 开发者团队的一员，他是一位培训师和顾问。

Gavin King 是 Hibernate 项目的发起者以及最初的 Java 持久化专家组(JSR 220)的一员。他还主导了 CDI 的标准化工作(JSR 299)。Gavin 目前正在创建名为 Ceylon 的新编程语言。

Gary Gregory 是 Rocket Software 的首席软件工程师，他致力于应用程序服务器和遗留系统的集成。他是 Manning 出版社 *JUnit in Action* 和 *Spring Batch in Action* 这两本书的合著者，并且是 Apache 软件基金会项目：Commons、HttpComponents、Logging Services 和 Xalan 的项目管理委员会的一员。

第 2 章 开启一个项目	17
2.1 Hibernate 介绍	17
2.2 使用 JPA 的 “Hello World”	18
2.2.1 配置一个持久化单元	18
2.2.2 编写一个持久化类	20
2.3 存储和检索数据	21
2.4 本章小结	26
第 3 章 数据模型和元数据	27
3.1 Cavenhampter 示例应用程序	28
3.1.1 一个分层结构	28
3.1.2 分析和业务域	29
3.1.3 Cavenhampter 数据库	30
3.2 实现数据模型	31
3.2.1 创建数据库表	31
3.2.2 创建及持久化元数据	32

4.1.1 创建数据库表	35
4.1.2 定义数据库表结构	36
4.1.3 区分实体和值类型	39
4.2 映射具有标识的实体	38
4.2.1 创建 Java 标识和标识类	38
4.2.2 第一个实体类映射	39
4.2.3 选择一个主键	60
4.2.4 配置持久化策略	61
4.2.5 创建持久化策略	62
4.3 实体映射选项	66
4.3.1 配置名称	66
4.3.2 配置名称生成策略	69
4.3.3 配置版本策略	69
4.3.4 配置一个实体映射到子查询	70
4.4 本章小结	71
第 5 章 映射数据类型	73
5.1 映射基本类型	73

封面插图

《Hibernate 实战(第2版)》的封面插图来自于伦敦老邦德街的 William Miller 于 1802 年 1 月 1 日出版的奥斯曼帝国服装图集。该图集的扉页已经丢失,并且我们至今都无法找到它的下落。本书的目录同时使用英语和法语来标识插图,每张插图都有创作它的两位艺术家的姓名,他们无疑会为自己的作品装饰到 200 年后的一本计算机编程书籍封面上而感到惊讶。

就像出现在我们封面上的其他插图一样,奥斯曼图集的图片生动活泼地展现了两个世纪以前丰富多彩的服饰风俗。它们引发了我们对那个时期以及所有其他历史时期的隔离和距离感。自那时起,衣着习惯已经改变了,当时如此丰富的地区多样性已经逐渐消失。如今从服饰很难区分不同大陆的居民。也许,尝试从乐观的角度来看,我们已经用文化和视觉上的多样性换来了更为多样化的个人生活——或者说是更丰富以及有趣的知识技术生活。

我们这些 Manning 出版社的同仁崇尚创造性、进取性,这个图集集中的图片使得两个世纪以前丰富多彩的地区生活跃然于纸上,以其作为图书封面会让计算机行业多一些趣味性。

Hibernate 是基于宽松 GNU GPL 许可的开源项目。Hibernate 的 www.hibernate.org 上提供了以源代码或二进制格式下载 Hibernate 包的指示。<http://www.manning.com/books/java-persistence-with-hibernate-second-edition> 这个页面提供了每注册之后可以获得的折扣,以及该论坛执行的规则有关的信息。

作者在线

购买《Hibernate 实战(第2版)》就可以免费访问由 Manning 出版社运营的社区网络论坛。在该论坛中你可以发表与本书有关的评论、提出技术问题,并且接受来自作者及其他用户的帮助。要访问该论坛并订阅它,请把你的网络浏览器导航到 www.manning.com/books/java-persistence-with-hibernate-second-edition。这个页面提供了每注册之后可以获得的折扣,以及该论坛执行的规则有关的信息。

Manning 向我们的读者承诺,将提供读者之间以及读者与作者之间可以近行交流讨论的场所。就作者而言,无法确保其程度的参与度,他们对于该论坛的贡献(无论是数量或质量)的,我们建议你尝试向作者提出其有一些挑战性的问题,以便他们对讨论内容感兴趣。

目 录

第 I 部分 ORM 入门

第 1 章 理解对象/关系持久化	1
1.1 持久化的定义	4
1.1.1 关系型数据库	4
1.1.2 理解 SQL	5
1.1.3 在 Java 中使用 SQL	5
1.2 范式不匹配	7
1.2.1 粒度问题	8
1.2.2 子类型问题	10
1.2.3 标识问题	11
1.2.4 与关联相关的问题	12
1.2.5 数据导航的问题	13
1.3 ORM 和 JPA	14
1.4 本章小结	15
第 2 章 开启一个项目	17
2.1 Hibernate 介绍	17
2.2 使用 JPA 的“Hello World”	18
2.2.1 配置一个持久化单元	18
2.2.2 编写一个持久化类	20
2.2.3 存储和加载消息	21
2.3 原生 Hibernate 配置	23
2.4 本章小结	26
第 3 章 域模型和元数据	27
3.1 CaveatEmptor 示例应用程序	28
3.1.1 一个分层架构	28
3.1.2 分析业务域	29
3.1.3 CaveatEmptor 域模型	30
3.2 实现域模型	31
3.2.1 处理关注点渗漏	31
3.2.2 透明及自动持久化	32

3.2.3 编写可持久化类	33
3.2.4 实现 POJO 关联	36
3.3 域模型元数据	39
3.3.1 基于注解的元数据	40
3.3.2 应用 Bean 验证规则	42
3.3.3 使用 XML 文件外部化元数据	45
3.3.4 在运行时访问元数据	48
3.4 本章小结	51

第 II 部分 映射策略

第 4 章 映射持久化类	55
4.1 理解实体和值类型	55
4.1.1 细粒度域模型	55
4.1.2 定义应用程序概念	56
4.1.3 区分实体和值类型	57
4.2 映射具有标识的实体	58
4.2.1 理解 Java 标识和相等性	58
4.2.2 第一个实体类和映射	59
4.2.3 选择一个主键	60
4.2.4 配置键生成器	61
4.2.5 标识符生成器策略	63
4.3 实体映射选项	66
4.3.1 控制名称	66
4.3.2 动态 SQL 生成	69
4.3.3 让实体不可变	69
4.3.4 将一个实体映射到子查询	70
4.4 本章小结	71
第 5 章 映射值类型	73
5.1 映射基本属性	73

5.1.1	重写基本属性的默认设置	74	7.1.2	创建和映射一个集合 属性	126
5.1.2	自定义属性访问	75	7.1.3	选择集合接口	127
5.1.3	使用派生属性	77	7.1.4	映射集	128
5.1.4	转换列值	77	7.1.5	映射标识符包	129
5.1.5	生成的以及默认的属性值	78	7.1.6	映射列表	130
5.1.6	时序属性	79	7.1.7	映射一个映射	132
5.1.7	映射枚举	80	7.1.8	排列和排序集合	132
5.2	映射可嵌入组件	80	7.2	组件集合	135
5.2.1	数据库架构	81	7.2.1	组件实例的相等性	136
5.2.2	让类可嵌入	81	7.2.2	组件集	138
5.2.3	重写嵌入属性	84	7.2.3	组件包	139
5.2.4	映射嵌套的可嵌入组件	85	7.2.4	组件值的映射	141
5.3	使用转换器映射 Java 和 SQL 类型	87	7.2.5	作为映射键的组件	142
5.3.1	内置类型	87	7.2.6	可嵌入组件中的集合	143
5.3.2	创建自定义 JPA 转换器	92	7.3	映射实体关联	144
5.3.3	使用 UserTypes 扩展 Hibernate	98	7.3.1	最简单的可能关联	145
5.4	本章小结	104	7.3.2	让其变成双向的	146
第 6 章	映射继承关系	105	7.3.3	级联状态	147
6.1	每个带有隐式多态的具体类 使用一个表	105	7.4	本章小结	153
6.2	每个带有联合的具体类使用 一个表	107	第 8 章	高级实体关联映射	155
6.3	每个类层次结构使用一个表	109	8.1	一对一关联	155
6.4	每个带有联结的子类使用 一个表	112	8.1.1	共享主键	156
6.5	混合继承策略	115	8.1.2	外主键生成器	158
6.6	可嵌入类的继承	117	8.1.3	使用一个外键联结列	161
6.7	选择一种策略	119	8.1.4	使用一个联结表	162
6.8	多态关联	120	8.2	一对多关联	164
6.8.1	多态多对一关联	121	8.2.1	考虑一对多包	164
6.8.2	多态集合	123	8.2.2	单向和双向列表映射	166
6.9	本章小结	124	8.2.3	具有联结表的可选 一对多	168
第 7 章	映射集合和实体关联	125	8.2.4	可嵌入类中的一对多 关联	169
7.1	集、包、列表及值类型 映射	125	8.3	多对多和三元关联	171
7.1.1	数据库架构	126	8.3.1	单向和双向多对多关联	172
			8.3.2	具有中间实体的多对多 关联	173
			8.3.3	具有组件的三元关联	177

8.4 具有映射的实体关联	180
8.4.1 具有属性键的一对多 关联	180
8.4.2 键/值三元关系	181
8.5 本章小结	182
第 9 章 复杂和遗留模式	185
9.1 改进数据库架构	186
9.1.1 添加辅助数据库对象	186
9.1.2 SQL 约束	189
9.1.3 创建索引	194
9.2 处理遗留键	195
9.2.1 映射一个自然主键	195
9.2.2 映射一个组合主键	196
9.2.3 组合主键中的外键	198
9.2.4 引用组合主键的外键	201
9.2.5 引用非主键的外键	202
9.3 将属性映射到辅助表	203
9.4 本章小结	204

第III部分 事务性数据处理

第 10 章 管理数据	207
10.1 持久化生命周期	207
10.1.1 实体实例状态	208
10.1.2 持久化上下文	209
10.2 EntityManager 接口	211
10.2.1 规范的工作单元	211
10.2.2 使数据持久化	212
10.2.3 检索和修改持久化 数据	213
10.2.4 得到一个引用	215
10.2.5 让数据变成瞬时的	216
10.2.6 刷新数据	217
10.2.7 复制数据	217
10.2.8 在持久化上下文中 缓存	218
10.2.9 刷新持久化上下文	220
10.3 处理分离的状态	221
10.3.1 分离实例的标识	221

10.3.2 实现相等性方法	223
10.3.3 分离实体实例	225
10.3.4 合并实体实例	226
10.4 本章小结	227

第 11 章 事务和并发

11.1 事务的要素	229
11.1.1 ACID 属性	230
11.1.2 数据库和系统事务	230
11.1.3 使用 JTA 的编程式 事务	230
11.1.4 处理异常	232
11.1.5 声明式事务分界	234
11.2 控制并发访问	234
11.2.1 理解数据库级别的 并发	235
11.2.2 乐观并发控制	239
11.2.3 显式悲观锁	245
11.2.4 避免死锁	248
11.3 非事务性数据访问	249
11.3.1 在自动提交模式中读取 数据	250
11.3.2 对修改进行排队	251
11.4 本章小结	253

第 12 章 抓取计划、策略和配置

文件	255
12.1 延迟加载和急加载	256
12.1.1 理解实体代理	256
12.1.2 延迟持久化集合	260
12.1.3 使用拦截进行延迟 加载	262
12.1.4 关联和集合的急加载	264
12.2 选择一个抓取策略	266
12.2.1 n+1 查询问题	266
12.2.2 笛卡尔积问题	267
12.2.3 批量预抓取数据	270
12.2.4 使用子查询预抓取集合	272
12.2.5 使用多个 SELECT 进行急 抓取	273

12.2.6	动态急抓取	274
12.3	使用抓取配置文件	275
12.3.1	声明 Hibernate 抓取配置 文件	276
12.3.2	使用实体图	277
12.4	本章小结	281
第 13 章	数据过滤	283
13.1	级联状态迁移	284
13.1.1	可用的级联选项	284
13.1.2	传递式分离与合并	285
13.1.3	级联刷新	287
13.1.4	级联复制	289
13.1.5	启用全局传递式 持久化	290
13.2	侦听和拦截事件	290
13.2.1	JPA 事件侦听器和 回调	291
13.2.2	实现 Hibernate 拦截器	294
13.2.3	核心事件系统	298
13.3	使用 Hibernate Envers 进行 审计和版本控制	299
13.3.1	启用审计日志	300
13.3.2	创建审计追踪	301
13.3.3	找出版本	301
13.3.4	访问历史数据	303
13.4	动态数据过滤器	305
13.4.1	定义动态过滤器	306
13.4.2	应用过滤器	306
13.4.3	启用过滤器	307
13.4.4	过滤集合访问	308
13.5	本章小结	309

第 IV 部分 编写查询

第 14 章	创建和执行查询	313
14.1	创建查询	314
14.1.1	JPA 查询接口	314
14.1.2	类型化查询结果	316
14.1.3	Hibernate 的查询接口	316

14.2	准备查询	318
14.2.1	防止 SQL 注入攻击	318
14.2.2	绑定命名参数	318
14.2.3	使用定位参数	320
14.2.4	对大结果集分页	320
14.3	执行查询	322
14.3.1	列示所有结果	322
14.3.2	得到单个结果	322
14.3.3	滚动数据库游标	323
14.3.4	遍历一个结果	325
14.4	命名和外部化查询	326
14.4.1	调用一个命名查询	326
14.4.2	在 XML 元数据中定义 查询	326
14.4.3	使用注解定义查询	327
14.4.4	程式化定义命名查询	328
14.5	查询提示	329
14.5.1	设置一个超时时长	330
14.5.2	设置刷新模式	330
14.5.3	设置只读模式	331
14.5.4	设置一个抓取大小	331
14.5.5	设置一个 SQL 注释	331
14.5.6	命名的查询提示	332
14.6	本章小结	333
第 15 章	查询语言	335
15.1	选择	336
15.1.1	指定别名和查询根	336
15.1.2	多态查询	337
15.2	限制	338
15.2.1	比较表达式	339
15.2.2	使用集合的表达式	344
15.2.3	调用函数	345
15.2.4	对查询结果排序	347
15.3	投影	348
15.3.1	实体和标量值的投影	348
15.3.2	使用动态实例化	350
15.3.3	得到唯一结果	351
15.3.4	在投影中调用函数	352

15.3.5	聚合函数	354
15.3.6	分组	355
15.4	联结	357
15.4.1	使用 SQL 进行联结	357
15.4.2	JPA 中的联结选项	359
15.4.3	隐式关联联结	359
15.4.4	显式联结	361
15.4.5	使用联结进行动态 抓取	363
15.4.6	theta 风格的联结	366
15.4.7	比较标识符	367
15.5	子查询	369
15.5.1	相关与不相关的嵌套	369
15.5.2	量化	370
15.6	本章小结	371
第 16 章	高级查询选项	373
16.1	转换查询结果	373
16.1.1	返回一系列列表	374
16.1.2	返回一系列映射	375
16.1.3	将别名映射到 bean 属性	376
16.1.4	编写一个 ResultTransformer	376
16.2	过滤集合	377
16.3	Hibernate 条件查询 API	380
16.3.1	选择和排序	380
16.3.2	限制	381
16.3.3	投影和聚合	382
16.3.4	联结	383
16.3.5	子查询	385
16.3.6	示例查询	385
16.4	本章小结	387
第 17 章	自定义 SQL	389
17.1	回退到 JDBC	390
17.2	映射 SQL 查询结果	391
17.2.1	使用 SQL 查询进行 投影	392
17.2.2	映射到一个实体类	393

17.2.3	自定义结果映射	395
17.2.4	外部化原生查询	406
17.3	自定义 CRUD 操作	410
17.3.1	启用自定义加载器	410
17.3.2	自定义创建、更新和 删除	411
17.3.3	自定义集合操作	412
17.3.4	在自定义加载器中急 抓取	414
17.4	调用存储过程	417
17.4.1	返回一个结果集	418
17.4.2	返回多个结果以及更新 计数	419
17.4.3	设置输入和输出参数	421
17.4.4	返回一个游标	423
17.5	将存储过程用于 CRUD	425
17.5.1	自定义一个具有过程的 加载器	425
17.5.2	用于 CUD 的过程	426
17.6	本章小结	428

第 V 部分 构建应用程序

第 18 章	设计客户端/服务器应用 程序	431
18.1	创建持久化层	432
18.1.1	一种通用的数据访问对象 模式	433
18.1.2	实现通用接口	434
18.1.3	实现实体 DAO	436
18.1.4	测试持久化层	438
18.2	构建一个无状态服务器	439
18.2.1	编辑一个拍卖商品	440
18.2.2	放置出价	442
18.2.3	分析无状态应用程序	446
18.3	构建一个状态服务器	447
18.3.1	编辑一个拍卖商品	448
18.3.2	分析状态性应用程序	452
18.4	本章小结	454

第 19 章 构建 Web 应用程序	455	第 20 章 扩展 Hibernate	487
19.1 集成 JPA 与 CDI	455	20.1 大量和批量处理	487
19.1.1 生成一个 EntityManager	456	20.1.1 JPQL 和条件中的大批量 语句	488
19.1.2 将 EntityManager 与事务 联结起来	457	20.1.2 SQL 中的大批量语句	492
19.1.3 注入一个 EntityManager	458	20.1.3 批处理	493
19.2 数据的分页和排序	459	20.1.4 Hibernate StatelessSession 接口	496
19.2.1 偏移量分页与搜寻分页 对比	459	20.2 缓存数据	498
19.2.2 在持久化层中分页	461	20.2.1 Hibernate 共享的缓存 架构	498
19.2.3 逐页查询	466	20.2.2 配置共享缓存	502
19.3 构建 JSF 应用程序	468	20.2.3 启用实体和集合缓存	503
19.3.1 请求作用域服务	468	20.2.4 测试共享缓存	506
19.3.2 会话作用域服务	471	20.2.5 设置缓存模式	508
19.4 序列化域模型数据	478	20.2.6 控制共享缓存	509
19.4.1 编写一个 JAX-RS 服务	479	20.2.7 查询结果缓存	510
19.4.2 应用 JAXB 映射	480	20.3 本章小结	512
19.4.3 序列化 Hibernate 代理	482		
19.5 本章小结	485		

第 I 部分

ORM 入门

第 I 部分将介绍为何对象持久化是一个十分复杂的主题以及可以在实践中应用什么样的解决方案。第 1 章将介绍对象/关系范式不匹配和应对这一情形的几种策略，最重要的一种策略就是对象/关系映射(ORM)。在第 2 章中，我们将通过教程引导你逐步使用 Hibernate 和 Java 持久化——你要实现和测试一个“Hello World”示例。对此做好准备之后，在第 3 章中就可以学习如何在 Java 中设计和实现复杂业务领域模型，以及当前有哪些可用的映射元数据选项。

阅读完本书的这一部分后，你将理解为何需要 ORM 以及 Hibernate 和 Java 持久化是如何在实践中发挥作用的。你要编写自己的第一个小项目，并且要准备好承担处理更为复杂问题的责任。你还将理解如何将现实环境的业务实体作为一个 Java 域模型来实现，以及你更愿意用何种格式来处理 ORM 元数据。

广泛应用的持久化解决方案包括：数据库中的存储、通过一下 Web 应用程序框架(JavaServer Faces 对比 Struts，或者 Spring 对比 J2EE 容器框架(Swing 对比 SWT)或者模板引擎(OSF 对比 Thymeleaf) 等) 的持久化策略。每个持久化策略都有其优缺点和劣势，但它们都是基于相同的原理来用和总体策略。遗憾的是，持久化技术并非如此，简而言之，我们会看到一些针对相同问题的不同解决方案。

持久化一直是 Java 程序员们争论的焦点话题。这是一个已经被 SQL 处理存储过程这样的扩展所解决的问题，还是应该使用各种 EJB 这样的特殊 Java 组件框架来处理的问题？我们应该在 SQL 语言之上手工编写最原始的 CRUD(创建、读取、更新、删除)操作，还是应该让这一工作交给数据库？每个数据库管理系统都有其自己的 SQL 方言，我们要如何实现可移植性呢？数据库技术会抛弃 SQL 并采用一种面向对象数据库系统或者 NoSQL 系统这样的不同数据库系统吗？争论可能永远不会结束，但称为对象/关系映射(ORM)的解决方案现在已经得到广泛认可，这在很大程度上要归功于 Hibernate 的革新。它是一种开源 ORM 框架实现。

在能够开始使用 Hibernate 之前，我们需要理解对象持久化与 ORM 的核心问题。这一章将帮助你为可移植使用 Hibernate 这样的工具以及遵循 Java 持久化 API(JPA)这样的规范。

首先我们将定义面向对象应用程序中的持久化数据资源，并探讨 SQL、JDBC 和 Java 的关系。Hibernate 构建在这些基础技术和标准上，然后我们要探讨所谓的对象/关

理解对象/关系

持久化

本章内容简介:

- 在 Java 应用程序中使用 SQL 数据库进行持久化
- 对象/关系范式不匹配
- 介绍 ORM、JPA 与 Hibernate

本书主要围绕 Hibernate 进行讲解;我们的介绍重点是使用 Hibernate 作为 Java 持久化 API 的提供者。我们将讲解基础和高级特性,并且介绍使用 Java 持久化开发新应用程序的一些方式。通常,这些建议并非特定于 Hibernate 的。有时,处理在 Hibernate 上下文中解释的持久化数据时,它们只是我们自己对于处理任务的最佳方式的理解。

管理持久化数据的方法已成为我们从事的每个软件项目中的一项关键设计决策。鉴于持久化数据并非 Java 应用程序中一种新的或者不寻常的需求,你会期望能够在类似的、已广泛应用的持久化解决方案之间轻易做出选择。思考一下 Web 应用程序框架(JavaServer Faces 对比 Struts,再对比 GWT)、GUI 组件框架(Swing 对比 SWT)或者模板引擎(JSP 对比 Thymeleaf)。每个相互竞争的解决方案都有各种优势和劣势,但它们都基于相同的适用范围和总体举措。遗憾的是,持久化技术并非如此,就其而言,我们会看到一些针对相同问题的迥异解决方案。

持久化一直是 Java 社区中的热点争论话题。这是一个已经被 SQL 及像存储过程这样的扩展所解决的问题,还是一个必须由像各种 EJB 这样的特殊 Java 组件模型来处理的更普遍问题?我们应该在 SQL 和 JDBC 中手工编写最原始的 CRUD(创建、读取、更新、删除)操作,还是应该让这一工作自动化?如果每个数据库管理系统都有其自己的 SQL 方言,我们要如何实现可移植性呢?我们应该完全抛弃 SQL 并采用一种像对象数据库系统或者 NoSQL 系统这样的不同数据库技术吗?争论可能永远不会结束,但称为对象/关系映射(ORM)的解决方案现在已经得到广泛认可,这在很大程度上要归功于 Hibernate 的革新,它是一种开源 ORM 服务实现。

在能够开始使用 Hibernate 之前,我们需要理解对象持久化和 ORM 的核心问题。这一章将阐明你为何需要像 Hibernate 这样的工具以及像 Java 持久化 API(JPA)这样的规范。

首先我们定义面向对象应用程序上下文中的持久化数据管理,并探讨 SQL、JDBC 和 Java 的关系, Hibernate 就构建在这些基础技术和标准上。然后我们要探讨所谓的对象/关

系范式不匹配,以及我们在使用 SQL 数据库的面向对象软件开发中所遇到的普遍问题。这些问题清晰地表明,我们需要工具和模式来将必须花费在应用程序中持久化相关代码上的时间最小化。

学习 Hibernate 的最佳方式并非一定是线性的。我们理解你可能希望立即尝试使用 Hibernate。如果这是你想要继续学习的方式,则可以跳到第 2 章并用“Hello World”示例建立一个项目。我们建议你在阅读本书过程中的某一刻回顾本章,这样你将做好准备,并获得阅读其余内容需要的所有背景概念。

1.1 持久化的定义

几乎所有的应用程序都需要持久化数据。持久化是应用程序开发中的基本概念之一。如果一个信息系统在关机时不保存数据,则该系统就没什么实际用途。对象持久化意味着个体对象可以比应用程序进程存在得更久;它们可以保存到数据存储并在以后的某个时点重建。当我们谈论 Java 中的持久化时,通常谈论的是使用 SQL 在一个数据库中映射和存储对象实例。首先我们简要介绍一下该技术以及如何在 Java 中使用它。了解这一信息基础,随后我们就可以继续探讨持久化以及如何在面向对象应用程序中实现它。

1.1.1 关系型数据库

就像大多数其他软件工程师一样,你可能已经使用过 SQL 和关系型数据库;许多软件工程师每天都在处理这样的系统。关系型数据库管理系统具有基于 SQL 的应用编程接口;因此,我们将如今的关系型数据库产品称为 SQL 数据库管理系统(DBMS),或者我们在谈论特定系统时将其称为 SQL 数据库。

关系技术是人们所熟知的技术,而这也成为许多组织选择它的充足理由。但要说它是唯一的理由,恐怕就不太尊重事实了。关系型数据库的应用已经根深蒂固了,因为它们对于数据管理来说是一种极具灵活性并且强健的方式。归因于对关系数据模型的深入研究的理论基础,关系型数据库可以保障和保护所存储数据的完整性,这是其众多可取特性之一。你可能对 E.F. Codd 于四十多年前提出的关系模型很熟悉,即 *A Relational Model of Data for Large Shared Data Banks*(Codd, 1970 年)。有一篇最近的摘要值得一读,其中专注于 SQL,这就是 C. J. Date 的 *SQL and Relational Theory*(Date, 2009 年)。

关系型 DBMS 并非特定于 Java,也不是特定于某特殊应用程序的一个 SQL 数据库。这一重要原则称为数据独立性。换句话说,我们怎么强调如下重要事实都不过分:数据的生命周期比任何应用程序都要长久。关系技术提供了在不同应用程序或者同一整体系统的不同部分(比如数据录入应用程序和报告应用程序)之间共享数据的方法。关系技术是许多异构系统和技术平台的通用标准。因此,关系数据模型通常是业务实体的常用企业级表示方式的基础。

在我们更详细地探究 SQL 数据库的实践方面之前,必须提及一个重要问题:即便作为关系型推向市场,但一个仅提供 SQL 数据语言接口的数据库系统并非真正的关系型数据库,并且在许多方面来说甚至与其原始概念都不相符。这自然而然会导致混淆。SQL 从业

人员抱怨关系数据模型在 SQL 语言方面的缺陷,同时关系数据管理专家则指责 SQL 标准对于关系模型和理念来说是一种弱实现。应用程序工程师被夹在当中,承担着交付某些有效成果的责任。在本书中,我们通篇着眼于强调这个问题的一些重要且意义重大的方面,但通常我们会关注实践部分。如果有兴趣了解更多的背景资料,我们强烈推荐 Fabian Pascal 所著的 *Practical Issues in Database Management: A Reference for the Thinking Practitioner* (Pascal, 2000 年)以及 Chris Date 所著的 *An Introduction to Database Systems* (Date, 2003 年),可以从了解(关系型)数据库系统的理论、概念以及理念。后一本书是一本优秀的参考书(它很厚重),从中可以找到在数据库和数据管理方面可能遇到的所有问题的答案。

1.1.2 理解 SQL

要有效使用 Hibernate,首先必须充分理解关系模型和 SQL。需要理解关系模型以及像保障数据完整性的规范化这样的主题,并且需要使用你的 SQL 知识来调优 Hibernate 应用程序的性能。Hibernate 能自动处理许多重复性的编码任务,但如果希望利用现代 SQL 数据库的完整功能,则必须扩充你的持久化技术知识而不是局限于 Hibernate 本身。要深入研究,可以参阅本书结尾处所列的参考书目。

你很可能已经有多年的 SQL 使用经验,并且熟悉用这一语言编写的基本操作以及语句。但从我们自身的经验来说,有时候 SQL 仍然难以记住,并且一些术语的使用各不相同。

我们来了解一些本书中用到的 SQL 术语。在 DBMS 的目录中创建、修改和删除像表和约束这样的构件时,你要将 SQL 用作一种数据定义语言(DDL)。准备好这一模式之后,可将 SQL 用作一种数据操作语言(DML)来执行对于数据的操作,其中包括插入、更新和删除。你要通过执行使用限制、投影和笛卡尔积的查询来检索数据。为得到有效的报告,你要在必要时使用 SQL 进行数据联结、聚合和分组。你甚至可在语句内部相互嵌套 SQL 语句——一项使用子查询的技术。当你的业务需求发生变化时,必须在数据存储之后再次使用 DDL 语句修改数据库模式;这称为架构演化。

如果使用 SQL 的经验丰富,并且希望了解更多与优化和 SQL 执行方式有关的内容,则可以参阅 Dan Tow 所著的优秀作品 *SQL Tuning* (Tow, 2003 年)。要从如何避免使用 SQL 的角度来研究 SQL 实践的方面, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming* (Karwin, 2010 年)一书就是不错的资源。

尽管 SQL 数据库是 ORM 的一个部分,但另一部分自然由你的 Java 应用程序中的数据构成,它们需要持久化到数据库并从中加载。

1.1.3 在 Java 中使用 SQL

当在 Java 应用程序中使用 SQL 数据库时,是通过 Java 数据库连接(JDBC)API 将 SQL 语句分发到数据库的。无论该 SQL 是手动编写并嵌入到 Java 代码中,还是由 Java 代码在运行时生成,你都要在准备查询参数、执行查询、滚动浏览查询结果、从结果集中检索值等操作时使用 JDBC API 来绑定实参。这些是低层次的数据访问任务;作为应用程序工程师,我们更关注需要这一数据访问操作的业务问题。我们真正想要编写的是保存和获取类

实例的代码，让我们从这些低层次的单调乏味的工作中解放出来。

由于这些数据访问任务通常都很枯燥无味，因此我们不得不进行思考，关系数据模型以及(尤其是)SQL 真的是用于面向对象应用程序中持久化的合适选择吗？我们可以十分肯定地回答这个问题：是的！SQL 数据库支配计算行业的原因有很多——关系型数据库管理系统是唯一行之有效的通用数据管理技术，并且它们几乎一直是 Java 项目中的必备条件。

注意，我们并不是在宣称关系技术总是最佳的解决方案。有许多数据管理的需求使得采用一种完全不同的方法成为必要条件。例如，互联网规模的分布式系统(网络搜索引擎、内容分发网络、点对点共享、即时通信)必须应对异常庞大的事务量。许多这些系统都不需要在一次数据更新完成之后，所有的进程得到相同的更新后数据(强事务一致性)。用户可能会对弱一致性感到满意；在一次更新完成之后，到所有进程得到更新后数据之前，可能会存在不一致的窗口期。某些科学的应用程序会处理庞大但非常专业的数据集。这样的系统及其独特的挑战往往同等地要求唯一性，并且通常需要定制的持久化解决方案。像兼容 ACID 的事务型 SQL 数据库、JDBC 以及 Hibernate 这样的通用数据管理工具只是扮演了次要角色而已。

互联网规模的关系型系统

要理解为何关系型系统以及与之相关联的数据完整性保障难以扩展，我们建议你首先要熟悉 CAP 法则。根据这一法则，一个分布式系统将无法在同一时间保持一致性、可用性以及针对分区故障的容错性。

一个系统可能可以确保所有节点在同一时刻得到相同数据，并且该数据的读写请求总是会得到响应。但是当该系统的一部分因为主机、网络或者数据中心问题出现故障时，你就必须要么放弃强一致性(线性一致性)，要么放弃 100% 的可用性。实际上，这意味着需要一条检测分区故障并且将一致性或者可用性恢复到某种程度的策略(例如，通过让系统的某部分暂时不可用来让后台进行数据同步)。通常这取决于强一致性对数据、用户或者操作是否是必要条件。

要了解旨在能轻易扩展的关系型 DBMS，可以看一下 VoltDB(www.voltdb.com)和 Nuodb(www.nuodb.com)。另一篇值得一读的文章是 *F1-The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business*(Shute, 2012 年)，其中介绍了谷歌如何扩展其用于广告业务的最重要数据库，以及为何该数据库是关系型/SQL。

在本书中，我们将思考数据存储以及在使用域模型的面向对象应用程序上下文中共享数据的问题。作为直接处理 `java.sql.ResultSet` 的行与列的替代方式，应用程序的业务逻辑要与特定于应用程序的面向对象域模型进行交互。例如，如果一个在线拍卖系统的 SQL 数据库模式具有 ITEM 和 BID 表，则该 Java 应用程序就要定义 Item 和 Bid 类。应用程序不再使用 ResultSet API 读取和写入特定行与列的值，而是加载和存储 Item 与 Bid 类的实例。

因此，在运行时应用程序会操作这些类的实例。一个 Bid 的每个实例都有对一个拍卖 Item 的引用，而每个 Item 都可以具有一组对 Bid 实例的引用集合。业务逻辑不是在数据库中(作为一个 SQL 存储过程)执行的；它是在 Java 中实现并且在应用层中执行的。这使得业务逻辑可使用复杂的面向对象的概念，比如继承和多态。例如，可以使用众所周知的设计

模式，比如策略模式、中介模式以及组合模式(参阅 *Design Patterns: Elements of Reusable Object-Oriented Software*[Gamma, 1995 年])，这些模式都依赖于多态方法调用。

值得注意的是：并非所有的 Java 应用程序都是如此设计的，它们也不应完全如此设计。没有域模型的简单应用程序可能会好得多。如果要完全满足你的需要，则可以使用 JDBC `ResultSet`。调用现有存储过程，还要读取它们的 SQL 结果集。许多应用程序都需要执行修改大规模数据集的程序，其大小可能接近于所有的数据。你可能会使用普通 SQL 查询来实现一些报告功能并将结果直接呈现在界面上。SQL 和 JDBC API 对于处理表格化数据表示来说极其有效，并且 JDBC 的行集合使得 CRUD 操作更容易。使用这样的持久化数据表示是简单且易于理解的。

不过，对于具有有效业务逻辑的应用程序来说，域模型方法有助于显著提高代码的可重用性和可维护性。实际上，这两种策略都是常用且必需的。

几十年来，开发人员一直在谈论范式不匹配。这一不匹配解释了为何每个企业项目都要在持久化相关的问题上耗费如此大的精力。这里的范式所指的是对象建模和关系建模，或者更具体来说，就是面向对象编程和 SQL。

有了这一认知，你就可以开始查看一些问题了——有些很好理解，有些不那么容易理解——即一个结合了这两种数据表示的应用程序必须解决：一个面向对象的域模型和一个持久化关系模型。我们进一步探讨一下所谓的范式不匹配。

1.2 范式不匹配

对象/关系范式不匹配可以分成几个部分，我们将逐一进行阐释。我们从一个毫无问题的简单示例开始探讨。当在此基础上进行构建时，你将看到不匹配开始显现。

假定你必须设计和实现一个在线电子商务应用程序。在这个应用程序中，需要一个类来表示与系统用户有关的信息，并且需要另一个类来表示与该用户的账单明细有关的信息，如图 1-1 所示。

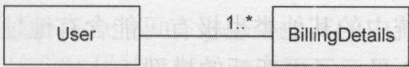


图 1-1 User 和 BillingDetails 实体的简单 UML 图

在图 1-1 中，可以看到一个 User 具有许多 BillingDetails。可以从两个方向浏览类之间的关系；这意味着可以遍历集合或者调用方法来得到关系的另一端。表示这些实体的类可能极其简单：

```
public class User {
    String username;
    String address;
    Set billingDetails;

    // Accessor methods (getter/setter), business methods, etc.
}

public class BillingDetails {
    String account;
```



```
String bankname;  
User user;  
  
// Accessor methods (getter/setter), business methods, etc.  
}
```

注意，你关注的只是与持久化有关的实体状态，因此我们省略了属性访问函数和业务方法的实现，比如 `getUsername()` 或 `billAuction()`。

很容易给出这个例子的 SQL 架构设计：

```
create table USERS (  
    USERNAME varchar(15) not null primary key,  
    ADDRESS varchar(255) not null  
);  
  
create table BILLINGDETAILS (  
    ACCOUNT varchar(15) not null primary key,  
    BANKNAME varchar(255) not null,  
    USERNAME varchar(15) not null,  
    foreign key (USERNAME) references USERS  
);
```

`BILLINGDETAILS` 中的外键约束列 `USERNAME` 表示了这两个实体之间的关系。对于这一简单域模型而言，几乎看不出对象/关系不匹配；编写 JDBC 代码来插入、更新和删除与用户及账单明细有关的信息非常简单。

现在让我们看看在考虑更现实一点的事情时会发生什么。在你将更多实体和实体关系添加到应用程序中时，范式不匹配就会浮现出来。

1.2.1 粒度问题

当前实现中最显而易见的问题就是，你将地址设计成了一个简单 `String` 值。在大多数系统中，分别保存街道、城市、州、国家和邮编信息是必要的。当然，可以将这些属性直接添加到 `User` 类，但由于系统中的其他类也极有可能含有地址信息，因此创建一个 `Address` 类是更加合理的举措。图 1-2 显示了更新后的模型。

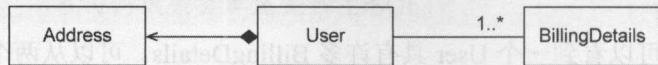


图 1-2 User 有一个 Address

还应该添加一个 `ADDRESS` 表吗？没有必要；通常的做法是将地址信息保存到 `USERS` 表中单独的列里面。这一设计可能会执行得更好，因为如果希望在单个查询中检索用户和地址的话，则不需要表联结。最佳解决方案可能是创建一个新的 SQL 数据类型来表示地址，并在 `USERS` 表中添加该新类型的单个列，而非添加几个新列。

可以选择添加几个列或者（一种新 SQL 数据类型的）单个列。这明显是一个粒度问题。总之，粒度指的是你正在使用的类型的相对大小。

我们回到该示例。将一个新的数据类型添加到数据库目录，以便在单个列中存储 `Address` Java 实例，这听起来是最佳方法：

```
create table USERS (  
    USERNAME varchar(15) not null primary key,  
    ADDRESS address not null  
);
```

Java 中的一个新 Address 类型(类)和新的 ADDRESS SQL 数据类型应该保证互操作性。不过,如果检查如今 SQL 数据库管理系统中对用户定义数据类型(UDT)的支持,你就会发现各种问题。

UDT 支持是所谓的对传统 SQL 的众多对象-关系扩展之一。单独来看,这个术语会令人困惑,因为它意味着数据库管理系统具有(或者说应该支持)一种完备的数据类型系统——如果某人推销给你一套可以用关系形式处理数据的系统,你会认为这是理所当然的事情。遗憾的是,UDT 支持是大多数 SQL DBMS 的一个略微令人费解的特性,且肯定无法在不同的产品之间移植。此外,SQL 标准虽然支持用户定义的数据类型,但其效果却很糟糕。

这一局限并非关系数据模型的问题。可以将如此重要的一项功能没有标准化的失误归咎于 20 世纪 90 年代中期供应商之间的对象-关系数据库之争。如今,大多数工程师都接受了 SQL 产品具有受限类型系统的局面——没人再提出质疑了。即便你的 SQL DBMS 中有一个完备的 UDT 系统,你仍然可能会重复类型的声明,在 Java 中编写新类型并在 SQL 中再次编写。遗憾的是,对于为 Java 领域找到一种更好的解决方案的尝试,比如 SQLJ,目前还没有太多的进展。DBMS 产品很少支持直接在数据库上部署和执行 Java 类,就算存在这样的支持,通常也会限制在非常基础的功能上,并且在日常使用中较为复杂。

基于这些以及其他原因,在一个 SQL 数据库中使用 UDT 或者 Java 类型,目前还不是业内普遍采用的做法,并且你不太可能遇到一个广泛使用 UDT 的遗留架构。因此你也不能也不会将新的 Address 类的实例存储在具有与 Java 层面相同数据类型的单个新列中。

这个问题的实用解决方案是使用几个供应商定义的内置 SQL 类型的列(比如布尔、数值和字符串数据类型)。你通常会将 USERS 表定义为:

```
create table USERS (  
    USERNAME varchar(15) not null primary key,  
    ADDRESS_STREET varchar(255) not null,  
    ADDRESS_ZIPCODE varchar(5) not null,  
    ADDRESS_CITY varchar(255) not null  
);
```

Java 域模型中的类具有各种各样的粒度级别:从像 User 这样的粗粒度实体类,到像 Address 这样的细粒度类,再到扩展 AbstractNumericZipCode 的简单 SwissZipCode(或者想要的任何抽象级别)。相较而言,SQL 数据库中只显示出两种粒度类型级别:由你创建的关系类型,比如 USERS 和 BILLINGDETAILS;以及内置数据类型,比如 VARCHAR、BIGINT 或 TIMESTAMP。

许多简单的持久化机制都无法识别这种不匹配,因而最终迫使在面向对象的模型上使用较不灵活的 SQL 产品表示,实际上使得模型变得扁平化了。

这证明了粒度问题并非特别难以解决。若非因为它存在于如此多的现有系统中这一事

实，我们甚至都可能不会谈论它。我们会在第 4.1 节中介绍这个问题的解决方案。

当我们考虑依赖继承的域模型时，就会出现一个更棘手且更值得关注的问题。这里的继承指的是面向对象设计的一个特性，你可用它来以新颖且有趣的方式对电子商务应用程序的用户进行结算。

1.2.2 子类型问题

在 Java 中，可以使用超类和子类来实现类型继承。为了揭示为何这会带来不匹配问题，让我们将其添加到你的电子商务应用程序中，以便现在不仅可以接受银行账户进行结算，也能接受信用卡和借记卡。在该模型中反映这一变化的最自然方式是将继承用于 `BillingDetails` 超类，以及使用几个具体的子类：`CreditCard`、`BankAccount` 等。这些子类中的每一个都定义了略微不同的数据(以及作用于这些数据上的完全不同的功能)。图 1-3 中的 UML 类图揭示了 this 模型。

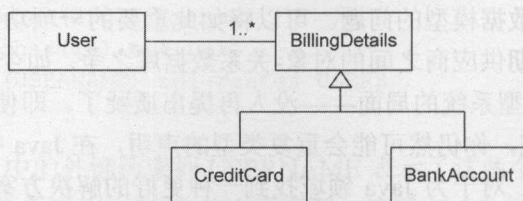


图 1-3 将继承用于不同的结算策略

要支持这一更新后的 Java 类结构，你必须进行哪些变更呢？可以创建一个扩展 `BILLINGDETAILS` 的 `CREDITCARD` 表吗？SQL 数据库产品一般不会实现表继承(甚或数据类型继承)，即便它们确实实现了表继承，也不会遵循一种标准语法，并且可能会让我们面临数据完整性问题(对可更新视图的有限完整性规则)。

继承的探讨还没有结束。一旦我们将继承引入模型之中，则会具有多态的可能性。

`User` 类与 `BillingDetails` 超类具有关联性，这是一种多态关联。在运行时，`User` 实体可能会引用 `BillingDetails` 任何子类的一个实例。类似地，你会期望能够编写引用 `BillingDetails` 类的多态查询，并且让该查询返回其子类的实例。

SQL 数据库还缺乏一种表示多态关联的明显方式(或者至少缺乏一种标准方式)。一个外键约束会准确地引用一张目标表；但定义一个引用多张表的外键并不简单。你必须编写一个程序性约束来强制实现这类完整性规则。

子类型这一不匹配的结果在于，模型中的继承结构必须持久化到一个不提供继承机制的 SQL 数据库中。在第 6 章中，我们将探讨像 Hibernate 这样的 ORM 解决方案如何解决将类层次结构持久化到 SQL 数据库的一张或多张表，以及如何实现多态行为。幸运的是，如今在社区中这个问题很容易理解，并且大多数解决方案差不多都支持相同的功能。

对象/关系不匹配问题的下一个方面是对象标识问题。你可能已经注意到，我们将 `USERNAME` 定义为 `USERS` 表主键的示例。这是一种好选择吗？你要如何在 Java 中处理相同的对象呢？

1.2.3 标识问题

尽管标识问题最初可能并不明显，但你将经常在日益增长和扩展的电子商务系统中遇到它，比如当需要检查两个实例是否相同的时候。有三种方法来处理这个问题：两种在 Java 中，一种在 SQL 数据库中。正如所预料的那样，只要提供一些帮助，它们就能协同工作了。

Java 定义了两种不同的相同性概念：

- 实例标识(大致等同于内存位置，使用 `a == b` 进行检查)
- 实例相等性，通过 `equals()` 方法(也称为值相等)的实现进行判定

另一方面，数据库行的标识会表示为主键值的比较。正如你将会在第 10.1.2 节中所看到的，`equals()` 或者 `==` 都不会总是等同于主键值的比较。Java 中有几个不相等的实例同时表示数据库同一行的情况是很常见的——例如，在并发运行的应用程序线程中就是如此。此外，在为持久化类正确实现 `equals()` 并且在理解何时可能必须这么做的时候，会涉及一些不易察觉的困难。

我们用一个示例来探讨另一个与数据库标识有关的问题。在 `USERS` 的表定义中，`USERNAME` 是主键。遗憾的是，这一做法使得用户名称很难修改；你不仅需要更新 `USERS` 中的行，还需要更新(许多)`BILLINGDETAILS` 行中的外键值。要解决这个问题，我们在本书的后续内容中建议，无论何时你无法找到一个好的自然键，则使用代理键。我们还会探讨怎样得到一个好的主键。代理键是对应用程序用户不具有任何意义的主键列——换句话说，它是不会呈现给应用程序用户的键。其存在的唯一目的就是识别应用程序内部的数据。

例如，你可能会像这样修改自己的表定义：

```
create table USERS (  
    ID bigint not null primary key,  
    USERNAME varchar(15) not null unique,  
    ...  
);  
  
create table BILLINGDETAILS (  
    ID bigint not null primary key,  
    ACCOUNT varchar(15) not null,  
    BANKNAME varchar(255) not null,  
    USER_ID bigint not null,  
    foreign key (USER_ID) references USERS  
);
```

ID 列包含系统生成的值。这些列纯粹是为了便于使用数据模型而引入的，那么在 Java 域模型中应该(如果需要的话)如何表示它们呢？我们将在第 4.2 节中探讨这一问题，并且将找到一个使用 ORM 的解决方案。

在持久化的上下文中，标识与系统如何处理缓存和事务密切相关。不同的持久化解决方案选择了不同的策略，这已经成了一个混乱的领域。我们将在第 10.1 节中介绍所有这些值得关注的主题——并讲解它们是如何相关的。

到目前为止，你所设计的电子商务应用程序框架已经显露出使用映射粒度、子类型以及标识的范式不匹配问题。你已经差不多做好了转向该应用程序的其他部分的准备，但首先我们需要探讨关联这一重要概念：如何映射和处理实体之间的关系。数据库中的外键约

束就是你所需要的一切吗？

1.2.4 与关联相关的问题

在你的域模型中，关联表示了实体之间的关系。User、Address 和 BillingDetails 类都是相关的；但不同于 Address，BillingDetails 是自我独立的。BillingDetails 实例存储在其自己的表中。在任何对象持久化解决方案中，关联映射和实体关联的管理都是核心概念。

面向对象的语言会使用对象引用来表示关联；但在关系的领域中，外键约束列表示了一个关联，它带有一些键值副本。约束是保证关联完整性的规则。这两种机制之间有着实质性的区别。

对象引用本来就具有方向性；关联是从一个实例到另一个实例。它们都是指针。如果实例之间的关联应该在两个方向都能导航，那么你就必须定义该关联两次，在每个关联的类中定义一次。你已经在该域模型类中看到了这一点：

```
public class User {
    Set billingDetails;
}
public class BillingDetails {
    User user;
}
```

在特定方向中导航对于一个关系数据模型来说没有什么意义，因为可以使用联结和投影操作创建任意的数据关联。其挑战在于将一个完全开放、独立于数据使用应用程序的数据模型映射到一个依赖应用程序的导航模型——这个特定应用程序所需的关联约束视图。

Java 关联可以具有多对多的多样性。例如，可以像这样定义类：

```
public class User {
    Set billingDetails;
}
public class BillingDetails {
    Set users;
}
```

不过，在 BILLINGDETAILS 表上声明的外键是一个多对一的关联：每个银行账户都链接到一个特定用户。每个用户可能具有多个已链接的银行账户。

如果希望在一个 SQL 数据库中表示多对多关联，则必须引入一个新的表，通常称之为链接表。大多数情况下，这个表不会出现在域模型中的任何地方。对于这个示例来说，如果考虑将用户和结算信息之间的关系变成多对多关联，那么你就需要定义如下链接表：

```
create table USER_BILLINGDETAILS (
    USER_ID bigint,
    BILLINGDETAILS_ID bigint,
    primary key (USER_ID, BILLINGDETAILS_ID),
    foreign key (USER_ID) references USERS,
    foreign key (BILLINGDETAILS_ID) references BILLINGDETAILS
);
```

你不再需要 BILLINGDETAILS 表上的 USER_ID 外键列和约束了；这一附加表现在可以管理两个实体之间的链接。我们将在第7章中探讨关联和集合映射。

到目前为止，我们考虑过的问题主要都是结构性的：通过思考一个纯粹静态的系统视图，你就会发现它们。可能对象持久化中的大多数困难问题都是一个动态问题：如何在运行时访问数据。

1.2.5 数据导航的问题

在 Java 和关系型数据库中访问数据有着根本性区别。在 Java 中，当访问一个用户的结算信息时，可以调用 `someUser.getBillingDetails().iterator().next()` 或者类似的方法。这是访问面向对象数据最自然的方式，并且它通常描述为遍历对象网络。跟随类之间准备好的指针，可以从一个实例导航到另一个实例，甚至迭代集合。遗憾的是，这并非从 SQL 数据库检索数据的一种高效方式。

要提高数据访问代码的性能，唯一能做的最重要事情就是最小化对数据库的请求量。达成此目的最显而易见的方式就是尽可能减少 SQL 查询的数量(当然，紧接着第二步就要使用其他更为复杂的方式——比如大量的缓存)。

因此，使用 SQL 对关系数据进行有效访问通常需要在相关的表之间使用联结。检索数据时所涉及的表数量将确定可以在内存中导航的对象网络的深度。例如，如果需要检索一个 User 并且不关心该用户的结算信息，则可以编写下面这个简单的查询：

```
select * from USERS u where u.ID = 123
```

另一方面，如果需要检索一个 User，随后要访问每一个相关联的 BillingDetails 实例(比如说，列出该用户的所有银行账户)，则可以编写另一个不同的查询：

```
select * from USERS u
  left outer join BILLINGDETAILS bd
    on bd.USER_ID = u.ID
where u.ID = 123
```

要有效使用联结，检索初始实例时需要在开始导航对象网络之前就弄清楚计划访问该对象网络的哪一部分！不过要当心：如果检索的数据过多(也许超过你可能需要的量)，则是在浪费应用层的内存。大量的笛卡尔积结果集也可能会导致 SQL 数据库的不可用。想象一下这样的情况，不仅仅在一条查询中检索用户和银行账户，还在其中检索每个银行账户支付的所有订单、每个订单中的产品等。

任何对象持久化解决方案都名副其实地提供了只在 Java 代码中首先访问关联时才提取关联实例数据的功能。这称为延迟加载：仅在需要时检索数据。这一数据访问的零碎方式在 SQL 数据库的上下文中根本就没有效率，因为它需要为每一个访问的节点或者对象网络的集合执行一条语句。这就是令人担心的 $n+1$ 查询问题。

你在 Java 和关系型数据库内数据访问方式的这一不匹配问题可能是 Java 信息系统中唯一最常见的性能问题的来源。不过，尽管我们已经在建议将 `StringBuffer` 用于字符串连接的大量书籍和文章中得到过启示，但避免笛卡尔积和 $n+1$ 查询问题对于许多 Java 编程人

员来说仍旧是一个谜团(承认吧:你只是认为 `StringBuilder` 会比 `StringBuffer` 更好)。

Hibernate 为有效、透明地从数据库中将对象网络提取到访问它们的应用程序提供了完备的特性。我们将在第 12 章中探讨这些特性。

现在我们有相当多的一些对象/关系不匹配问题,并且根据你的经验,要找到解决方案会需要很大的代价(时间和精力)。本书将花费大量篇幅为这些问题提供一个完整的答案,并且揭示 ORM 作为一个可行解决方案的可能性。让我们开始简要了解 ORM、Java 持久化标准以及 Hibernate 项目吧。

1.3 ORM 和 JPA

简单来说,对象/关系映射就是使用描述应用程序类和 SQL 数据库架构之间映射的元数据,将 Java 应用程序中的对象自动(且透明)地持久化到 SQL 数据库中的表。实质上,ORM 是通过将数据从一种表示转换成另一种表示(可逆的操作)来发挥作用。在我们继续讲解之前,需要理解 Hibernate 不能做什么。

ORM 的一个设想优势是为开发人员屏蔽掉凌乱复杂的 SQL。这一观点认为,不能期望面向对象的开发人员很好地理解 SQL 或者关系型数据库,并且不知为何他们对 SQL 抱有抵触情绪。相比之下,我们相信 Java 开发人员必须足够熟悉(并且欣赏)关系建模和 SQL,以便使用 Hibernate。ORM 是那些已经以艰苦方式实现它的开发人员所使用的一种先进技术。要有效使用 Hibernate,你必须能够查看和解释它所生成的 SQL 语句,并且理解这些 SQL 语句带来的性能影响。

以下是 Hibernate 的一些好处:

- 生产效率——Hibernate 去除了大量繁重的工作(远超你的想象),并且让可以专注于业务问题。无论你喜欢哪种应用程序开发策略——自上而下,从一个域模型开始;或者自下而上,从一个现有数据库架构开始——Hibernate 与合适的工具一起使用将显著减少开发时长。
- 可维护性——使用 Hibernate 的自动化 ORM 将减少代码行数(LOC),让系统变得更易于理解,并且更易于重构。Hibernate 在域模型和 SQL 架构之间提供了一个缓冲,避免每个模型的微小变更影响到其他模型。
- 性能——尽管手动编码的持久化可能运行得更快,就像汇编代码要比 Java 代码运行得更快一样,但像 Hibernate 这样的自动化解决方案允许在任何时间使用许多的优化。这种情况的一个示例是应用层中高效且可轻易调优的缓存。这意味着开发人员可以耗费更多精力来手动优化剩下的一些真正瓶颈,而非过早地优化所有内容。
- 供应商无关性——Hibernate 可以有助于降低与绑定供应商相关的一些风险。即使你计划永不变更自己的 DBMS 产品,但支持若干不同 DBMS 的 ORM 工具会使得某种程度的可移植性成为可能。此外,DBMS 无关性对于工程师使用轻量级本地数据库进行开发,但将测试和生产环境部署在不同系统中的开发场景是有助处的。Java 开发人员很好地接受了使用 Hibernate 进行持久化的方法,并且标准 Java 持久化

API 也是遵循类似的原则来设计的。

在最近的 EJB 和 Java EE 规范中, JPA 变成了其中引入的简化形式的一个关键部分。我们应该事先就弄清楚, Java 持久化和 Hibernate 都不受限于 Java EE 环境; 它们对于持久化问题来说是通用的, 任何类型的 Java(或 Groovy、Scala)应用程序都可以使用它们。

JPA 规范给出如下定义:

- 用于指定映射元数据的一个设施——持久化类及其属性是如何与数据库模式关联的。JPA 极度依赖领域模型类中的 Java 注解, 但也可以在 XML 文件中编写映射。
- 在持久化类的实例上执行基础 CRUD 操作的 API, 最明显的就是用 `javax.persistence.EntityManager` 存储和加载数据。
- 用于指定引用类及其属性的查询的一种语言和 API。这一语言就是 Java 持久化查询语言(JPQL), 看起来类似于 SQL。标准化的 API 允许程式化创建标准查询, 而不需要字符串操作。
- 持久化引擎如何与事务性实例交互以执行脏检查、关联抓取以及其他优化功能。最新的 JPA 规范涵盖了一些基础缓存策略。

Hibernate 实现了 JPA 并支持所有标准化的映射、查询以及编程接口。

1.4 本章小结

- 有了对象持久化, 独立的对象就能比其应用程序进程存在得更久, 可以保存到数据存储, 并且在之后重新创建。当数据存储是一个基于 SQL 的关系型数据库管理系统时, 对象/关系不匹配就会产生影响。例如, 对象的网络不能保存到数据库表; 它必须分解并持久化到可移植的 SQL 数据类型列。这一问题的优秀解决方案是对对象/关系映射(ORM)。
- ORM 并非所有持久化任务的灵丹妙药; 其职责是为开发人员减轻 95% 的对象持久化工作, 比如编写具有许多表联结的复杂 SQL 语句, 并且从 JDBC 结果集复制值到对象或者对象图。
- 一个功能完备的 ORM 中间件解决方案可以提供数据库可移植性、像缓存这样的特定优化技术、以及其他在限定时间内不能用 SQL 和 JDBC 轻易手动编码的切实可行的功能。
- 未来可能会出现比 ORM 更好的解决方案。我们(和其他许多人)可能必须重新思考所熟知的与数据管理系统及其语言、持久化 API 标准和应用程序集成有关的一切内容。但如今的系统演化成为具有无缝面向对象集成的真正关系型数据库系统仍旧纯粹是一种推测而已。我们不能止步不前, 并且没有迹象表明所有这些问题将很快得到改善(一个价值数十亿美元的行业不会非常灵活)。ORM 是当前可用的最佳解决方案, 它为每天面对对象/关系不匹配问题的开发人员节省了时间。

开启一个项目

第 2 章

2

本章内容简介：

- Hibernate 项目概述
- 使用 Hibernate 和 Java 持久化实现 “Hello World”
- 配置和集成选项

在这一章中，我们将开始使用 Hibernate 和 Java 持久化对一个循序渐进的示例进行讲解。你将看到这两个持久化 API，并且了解如何从原生 Hibernate 或者标准化 JPA 的使用中获益。我们首先会通过使用一个基于 Hibernate 的直观 “Hello World” 应用程序为你提供指引。在开始编码之前，你必须决定在项目中使用哪个 Hibernate 模块。

2.1 Hibernate 介绍

Hibernate 是一个雄心勃勃的项目，旨在为 Java 中管理持久化数据的问题提供一套完整的解决方案。如今，Hibernate 不仅是一个 ORM 服务，还是一套扩展到远超出 ORM 的数据管理工具集。

Hibernate 项目套件包括以下部分：

- **Hibernate ORM**——Hibernate ORM 由一个核心、一个使用 SQL 数据持久化的基础服务以及一个原生专用 API 构成。Hibernate ORM 是若干其他项目的基础，并且是最早的 Hibernate 项目。可以独立使用 Hibernate ORM，不依赖任何框架或者任何具有所有 JDK 的特定运行时环境。它可以在每一个 Java EE/J2EE 应用程序服务器中运行，比如在 Swing 应用程序中运行、在简单服务端小程序容器中运行等。只要你能为 Hibernate 配置一个数据源，它就能运行。
- **Hibernate 实体管理器**——这是 Hibernate 的标准 Java 持久化 API 实现，一个可在 Hibernate ORM 之上堆叠的可选模块。当需要一个普通 Hibernate 接口甚或一个 JDBC 连接时，可以退回到 Hibernate。从各方面来说，Hibernate 的原生特性都是 JPA 持久化特性的一个超集。
- **Hibernate 验证器**——Hibernate 提供了 Bean 验证(JSR 303)规范的参考实现。不依赖其他 Hibernate 项目，该验证器为你的域模型(或任何其他)类提供了声明式验证。

- **Hibernate Envers**——Envers 专用于审计日志和在你的 SQL 数据库中保留数据的多个版本。这有助于你添加数据历史以及审计追踪你的应用程序，类似于可能已经熟悉的版本控制系统，如 Subversion 和 Git。
- **Hibernate 搜索**——Hibernate 搜索会在一个 Apache Lucene 数据库中保持域模型数据的最新索引。它可以使用一个强大且原生集成的 API 查询这个数据库。除了 Hibernate ORM 之外，许多项目都使用 Hibernate 搜索，以增加全文搜索功能。如果应用程序的用户界面中有一个自由文本搜索窗体，并且你希望让用户满意，则可以使用 Hibernate 搜索。本书不会介绍 Hibernate 搜索；可以在 Emmanuel Bernard 所著的 *Hibernate Search in Action* 一书(Bernard, 2008 年)中找到更多相关信息。
- **Hibernate OGM**——最新的 Hibernate 项目是对象/网格映射器，它为 NoSQL 解决方案提供了 JPA 支持，重用了 Hibernate 核心引擎，但将映射的实体持久化到面向键/值、文档或图形的数据存储之中。本书不会介绍 Hibernate OGM。

下面开始创建你的第一个 Hibernate 和 JPA 项目吧。

2.2 使用 JPA 的“Hello World”

在本节中，你要编写首个 Hibernate 应用程序，该程序会在数据库中存储一条“Hello World”消息，然后对其进行检索。首先我们要安装并配置 Hibernate。

我们使用 Apache Maven 作为项目构建工具，它也是本书中所有示例会用到的工具。声明 Hibernate 上的依赖：

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.0.0.Final</version>
</dependency>
```

hibernate-entitymanager 模块包括你要用到的其他模块上的传递依赖项，比如 hibernate-core 和 Java 持久化接口存根。

你在 JPA 中的起点是持久化单元。持久化单元就是一对具有一个数据库连接的域模型类映射，外加一些其他配置设置。每一个应用程序至少都有一个持久化单元；有些应用程序如果要同几个(逻辑上或物理的)数据库通信，还会具有几个持久化单元。因此，第一步是在你的应用程序配置中设置一个持久化单元。

2.2.1 配置一个持久化单元

用于持久化单元的标准配置文件位于 META-INF/persistence.xml 中的类路径上。为该“Hello World”应用程序创建以下配置文件：

路径: /model/src/main/resources/META-INF/persistence.xml

```
<persistence
```

```

version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence_2_1.xsd"

<persistence-unit name="HelloWorldPU"> ← ①配置持久化单元

    <jta-data-source>myDS</jta-data-source> ← ②数据库连接

    <class>org.jpwh.model.helloworld.Message</class> ← ③持久化类

    <exclude-unlisted-classes>true</exclude-unlisted-classes> ← ④为映射的
                                                                    类禁用扫描

    <properties>
        <property
            name="javax.persistence.schema-generation.database.action"
            value="drop-and-create"/>
        <property name="hibernate.format_sql" value="true"/> ← ⑦格式化 SQL
        <property name="hibernate.use_sql_comments" value="true"/>
    </properties>
</persistence-unit>
</persistence>

```

⑤ 设置属性

⑥ 删除/重建 SQL 架构

- ① persistence.xml 文件要配置至少一个持久化单元；每个单元必须具有唯一的名称。
- ② 每个持久化单元必须具有一个数据库连接。此处你委托了一个已有的 `java.sql.DataSource`。Hibernate 在启动时将使用 JNDI 查找通过名称找到数据源。
- ③ 持久化单元具有持久化的(映射的)类，你要在此处列出它们。
- ④ Hibernate 可以扫描你的类路径查找映射的类，并将它们自动添加到你的持久化单元。这一设置禁用了该功能。
- ⑤ 标准选项或特定于供应商的选项可以在持久化单元上设置为属性。任何标准属性都有 `javax.persistence` 名称前缀；Hibernate 的设置使用 `hibernate` 前缀。
- ⑥ 在 JPA 引擎启动时，它应该自动删除和重建数据库中的 SQL 架构。这适用于自动化测试，当希望为每次测试运行提供一个干净的数据库时尤为有用。

在日志中打印 SQL 时，让 Hibernate 很好地格式化该 SQL 并在该 SQL 字符串中生成注释，以便你知道为何 Hibernate 执行了该 SQL 语句。大多数应用程序都需要一个数据库连接池，其中具有用于运行环境的一定大小的空间和优化的阈值。你还要提供 DBMS 主机以及用于数据库连接的凭据。

记录 SQL

所有由 Hibernate 执行的 SQL 语句都可以进行记录——这是优化期间极为有用的一个工具。要记录 SQL，可以在 persistence.xml 中将 `hibernate.format_sql` 和 `hibernate.use_sql_comments` 属性设置为 true。这样就能让 Hibernate 格式化具有起因注释的 SQL 语句。然后，在你的日志配置中(它依赖于选择的日志实现)，将 `org.hibernate.SQL` 和 `org.hibernate.type`。

descriptor.sql.BasicBinder 类别设置为最佳调试级别。然后就会在日志输出中看到所有由 Hibernate 执行的 SQL 语句，其中包括预处理语句的绑定参数值。

对于“Hello World”应用程序，你要将数据库连接处理委托给一个 Java 事务 API(JTA) 提供程序，即开源的 Bitronix 项目。Bitronix 提供了具有一个托管 java.sql.DataSource 的连接池和所有 Java SE 环境中都有的标准 javax.transaction.UserTransaction API。Bitronix 将这些对象绑定到 JNDI 之中，而 Hibernate 会通过 JNDI 查找自动与 Bitronix 连接。详细设置 Bitronix 不在本书讨论范围之内；可以在 org.jpwh.env.TransactionManagerSetup 中找到我们示例的配置。

在“Hello World”应用程序中，你希望将消息存储在数据库中，并且从数据库中加载它们。Hibernate 应用程序会定义映射到数据库表的持久化类。你要根据自己的业务域分析来定义这些类；因此，它们就是一个域模型。这个示例由一个类及其映射构成。

我们来看看一个简单持久化类的相关代码、如何创建映射以及可以在 Hibernate 中使用持久化类的实例做什么事情。

2.2.2 编写一个持久化类

这个示例的目标是将消息存储在一个数据库中，并且检索它们显示出来。该应用程序具有一个简单持久化类 Message：

路径：/model/src/main/java/org.jpwh/model/helloworld/Message.java

```
package org.jpwh.model.helloworld;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

@Entity

public class Message {

@Id

@GeneratedValue

 private Long id;

 private String text;

 public String getText() {
 return text;
 }

 public void setText(String text) {
 this.text = text;
 }

}

← ①所需的@Entity

← ②所需的@Id

← ③启用自动 ID 生成

← ④映射属性

- ① 每个持久化实体类都必须至少具有@Entity 注解。Hibernate 会将这个类映射到名为 MESSAGE 的表。

- ❷ 每个持久化实体类都必须具有一个用 `@Id` 注解的标识符属性, Hibernate 会将这一属性映射到名为 ID 的列。
- ❸ 必须生成标识符值; 这个注解将启用 ID 的自动生成。
- ❹ 通常你要实现具有私有或受保护字段以及公共 `getter/setter` 方法对的持久化类的常规属性。Hibernate 会将这个属性映射到名为 TEXT 的列。

持久化类的标识符属性允许应用程序访问其数据库实体——一个持久化实例的主键值。如果 `Message` 的两个实例具有相同的标识符值, 则它们代表的是数据库中相同的行。

这个示例中的标识符属性使用了 `Long` 类型, 但这并非必需的。Hibernate 允许使用几乎任何标识符类型, 稍后将会介绍。

你可能已经注意到, `Message` 类的 `text` 属性具有 `JavaBeansstyle` 属性访问器方法。该类还具有一个不带任何参数的(默认)构造函数。该示例中所示的持久化类通常会查找像这样的函数。注意, 你不必实现任何特定接口或者扩展任何特殊超类。

`Message` 类的实例可以由 Hibernate 托管(使之持久化), 但不必非得这样。因为 `Message` 对象不实现任何特定持久化类或接口, 所以可以像其他任何 Java 类一样使用它:

```
Message msg = new Message();  
msg.setText("Hello!");  
System.out.println(msg.getText());
```

可能看起来这有点举重若轻了; 但实际上, 我们是在揭示一个区分 Hibernate 和某些其他持久化解决方案的重要特征。可以在任何执行上下文中使用持久化类——无需任何特殊容器。

你不必使用注解来映射持久化类。稍后我们将介绍其他映射选项, 比如 `JPA orm.xml` 映射文件和原生的 `hbm.xml` 映射文件, 以及什么时候它们会是比源注解更好的解决方案。

现在 `Message` 类已经准备完毕。可以在数据库中存储实例, 并且编写查询将它们再次加载到应用程序内存中。

2.2.3 存储和加载消息

本书的重点是讲解 Hibernate, 所以我们来保存一条新的 `Message` 到数据库中。首先需要有一个 `EntityManagerFactory` 与数据库通信。这个 API 代表了你的持久化单元; 大多数应用程序都具有一个 `EntityManagerFactory`, 用于每一个配置好的持久化单元:

路径: `/examples/src/est/java/org/jpwh/helloworld/HelloWorldJPA.java`

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("HelloWorldPU");
```

当启动后, 你的应用程序应该创建 `EntityManagerFactory`; 该工厂类是线程安全的, 并且应用程序中访问该数据库的所有代码都应该共享它。

现在可以在一个划定的单元(一个事务)中使用该数据库, 并且存储一条 `Message`:

路径: /examples/src/test/java/org/jpwh/helloworld/HelloWorldJPA.java

```
UserTransaction tx = TM.getUserTransaction(); ← ①访问 UserTransaction
tx.begin();

EntityManager em = emf.createEntityManager(); ← ②创建 EntityManager

Message message = new Message(); ← ③创建 Message
message.setText("Hello World!");

em.persist(message); ← ④让实例持久化

tx.commit(); ← ⑤提交事务
// INSERT into MESSAGE (ID, TEXT) values (1, 'Hello World!')
em.close(); ← ⑥关闭 EntityManager
```

- ① 获得对标准事务 API `UserTransaction` 的访问权, 并在此执行线程上开始执行一个事务。
- ② 通过创建一个 `EntityManager` 来开始一个与数据库的新会话。这是你用于所有持久化操作的上下文。
- ③ 创建该映射的域模型类 `Message` 的一个新实例, 并设置其 `text` 属性。
- ④ 使用你的持久化上下文登记该临时实例; 让其持久化。现在 `Hibernate` 就知道你打算存储该数据, 但它不必立即调用数据库。
- ⑤ 提交该事务。`Hibernate` 会自动检查持久化上下文并执行必要的 SQL INSERT 语句。
- ⑥ 如果创建一个 `EntityManager`, 就必须关闭它。

为帮助你理解 `Hibernate` 是如何工作的, 我们将在 SQL 语句出现时在源代码注释中显示自动生成和执行的 SQL 语句。`Hibernate` 在 `MESSAGE` 表中插入了一行, 其 ID 主键列具有一个自动生成的值以及 `TEXT` 值。

之后可以使用一条数据库查询加载该数据:

路径: /examples/src/test/java/org/jpwh/helloworld/HelloWorldJPA.java

```
UserTransaction tx = TM.getUserTransaction(); ← ①事务边界
tx.begin();

EntityManager em = emf.createEntityManager();

List<Message> messages = ← ②执行查询
    em.createQuery("select m from Message m").getResultList();
// SELECT * from MESSAGE
assertEquals(messages.size(), 1);
assertEquals(messages.get(0).getText(), "Hello World!");

messages.get(0).setText("Take me to your leader!"); ← ③变更属性值

tx.commit(); ← ④执行 UPDATE
// UPDATE MESSAGE set TEXT = 'Take me to your leader!' where ID = 1

em.close();
```


- ❶ 与数据库的每一次交互都应该在明确的事务边界内发生，即便只是读取数据。
- ❷ 执行一条从数据库中检索所有 Message 实例的查询。
- ❸ 可以变更一个属性的值。Hibernate 会自动检测此变更，因为所加载的 Message 仍旧附加在其加载到其中的持久化上下文中。
- ❹ 在提交时，Hibernate 会检查持久化上下文是否处于脏状态，并且自动执行 SQL UPDATE 来将内存中的状态与数据库中的状态同步。

你在这个示例中看到的查询语言并非 SQL，它是 Java 持久化查询语言(JPQL)。尽管在这一寻常的示例中从语法上讲没有什么差别，但该查询字符串中的 Message 并不引用数据库表名称，而是引用持久化类名。如果将该类映射到另一个表，则该查询仍旧可以运行。

另外，要注意 Hibernate 是如何检测该消息文本属性的变更并自动更新数据库的。这是运行中 JPA 的自动脏检查特性。当在一个事务中修改一个实例的状态时，该特性可不必再明确要求你的持久化管理器更新数据库。

现在你已经完成了首个 Hibernate 和 JPA 应用程序。可能你已经注意到，我们倾向于将示例编写为可执行测试，其中带有验证每次操作正确输出的断言。我们使用了本书中测试代码的所有示例，因此你(以及我们)可以确信它们能够正常运行。遗憾的是，这也意味着在启动该测试环境时需要多行代码来创建 EntityManagerFactory。我们试图保持测试的设置尽可能简单。可以在 org.jpwh.env.JPASetup 和 org.jpwh.env.JPATest 中找到该代码；将其用作编写自己的测试工具的起始点。

在处理更加现实的应用程序示例之前，我们快速浏览一下原生的 Hibernate 引导程序和配置 API。

2.3 原生 Hibernate 配置

尽管在 JPA 中基础的(和大量的)配置被标准化，但你无法使用 persistence.xml 中的属性访问 Hibernate 的所有配置特性。注意，大多数应用程序，即便是非常复杂的应用程序，都无需如此特殊的配置选项，因而也就不必访问这一节中所介绍的引导 API。如果不确定，则可以跳过这一节，后续当需要扩展 Hibernate 类型适配器、添加自定义 SQL 函数等时再进行阅读。

原生的 org.hibernate.SessionFactory 等同于标准化的 JPA EntityManagerFactory。每个应用程序通常有一个该 API，并且是带有数据库连接配置的相同类映射对。

Hibernate 的原生引导 API 划分为几个阶段，每个阶段都可以访问特定的配置部分。用其最简洁的形式构建一个像下面这样的 SessionFactory：

路径：/examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
SessionFactory sessionFactory = new MetadataSources(  
    new StandardServiceRegistryBuilder()  
        .configure("hibernate.cfg.xml").build()  
)  
    .buildMetadata().buildSessionFactory();
```

这样就可以从一个 Hibernate 配置文件中加载所有的设置。如果有一个现有的 Hibernate 项目，则很可能你的类路径上已经有这个文件。类似于 `persistence.xml`，这个配置文件包含数据库连接详情，以及持久化类和其他配置属性的列表。

我们来解构这一引导程序代码片段并更为详细地查看该 API。首先，创建一个 `ServiceRegistry`：

路径：/examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
StandardServiceRegistryBuilder serviceRegistryBuilder = ①生成器
    new StandardServiceRegistryBuilder();

serviceRegistryBuilder
    .applySetting("hibernate.connection.datasource", "myDS")
    .applySetting("hibernate.format_sql", "true")
    .applySetting("hibernate.use_sql_comments", "true")
    .applySetting("hibernate.hbm2ddl.auto", "create-drop");

ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
```

②配置服务注册表

① 这一生成器有助于你创建带有链接方法调用的不可变服务注册表。

② 通过应用设置来配置服务注册表。

如果希望外部化服务注册表配置，则可以使用 `StandardServiceRegistryBuilder#loadProperties(file)` 从类路径上的一个属性文件中加载设置。

在构建了 `ServiceRegistry` 并让其不可变之后，就可以转到下一个阶段了：告知 Hibernate 哪些持久化类是映射元数据的一部分。像下面这样配置该元数据源：

路径：/examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
MetadataSources metadataSources = new MetadataSources(serviceRegistry);
metadataSources.addAnnotatedClass(
    org.jpwh.model.helloworld.Message.class
);
// Add hbm.xml mapping files
// metadataSources.addFile(...);
// Read all hbm.xml mapping files from a JAR
// metadataSources.addJar(...)
```

①需要服务注册表

②将持久化类添加到元数据源

```
MetadataBuilder metadataBuilder = metadataSources.getMetadataBuilder();
```

① 这一生成器有助于你创建带有链接方法调用的不可变服务注册表。

② 通过应用设置来配置服务注册表。

`MetadataSources` API 具有用于添加映射源的许多方法；查阅 Javadoc 以获得更多信息。引导程序的下一阶段是构建 Hibernate 所需的所有元数据，以及从元数据源中获得的 `MetadataBuilder`。

然后可以查询该元数据以便与 Hibernate 的完整配置进行程式化交互，或者继续构建最终的 `SessionFactory`：

路径: /examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
Metadata metadata = metadataBuilder.build();
assertEquals(metadata.getEntityBindings().size(), 1);
SessionFactory sessionFactory = metadata.buildSessionFactory();
```

根据 SessionFactory 创建 EntityManagerFactory

在撰写本书时, Hibernate 还没有便利的 API 可以程式化地构建一个 EntityManagerFactory。可以将一个内部 API 用于此目的: org.hibernate.jpa.internal.EntityManagerFactoryImpl 具有一个接受 SessionFactory 的构造函数。

我们来看看通过使用 Hibernate 等效于 EntityManager 的原生 org.hibernate.Session 存储和加载消息, 这一配置是否能生效。可以使用 SessionFactory 创建一个 Session, 并且必须关闭它, 正如你必须关闭自己的 EntityManager 一样。

或者, 通过另一个 Hibernate 特性, 可以用 SessionFactory#getCurrentSession() 让 Hibernate 负责创建和关闭该 Session:

路径: /examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
UserTransaction tx = TM.getUserTransaction(); ← ❶访问 UserTransaction
tx.begin();

Session session =
    sessionFactory.getCurrentSession(); ← ❷获取 org.hibernate.Session

Message message = new Message();
message.setText("Hello World!");

session.persist(message); ← ❸Hibernate API 和 JPA 是类似的

tx.commit();
// INSERT into MESSAGE (ID, TEXT) values (1, 'Hello World!')
```

❹提交事务

- ❶ 获得对标准事务 API UserTransaction 的访问权, 并在此执行线程上开始执行一个事务。
- ❷ 无论何时在相同线程中调用 getCurrentSession(), 都会得到相同的 org.hibernate.Session。它自动绑定到运行中的事务上, 并且在该事务提交或回滚时关闭。
- ❸ 原生的 Hibernate API 非常类似于标准的 Java 持久化 API, 并且大多数方法都有相同的名称。
- ❹ Hibernate 会与数据库同步会话并在提交绑定的事务时自动关闭“当前”会话。

用简洁的代码访问当前 Session 结果:

路径: /examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
UserTransaction tx = TM.getUserTransaction();
tx.begin();
```



```
List<Message> messages =  
    sessionFactory.getCurrentSession().createCriteria(  
        Message.class  
    ).list();  
// SELECT * from MESSAGE  
  
assertEquals(messages.size(), 1);  
assertEquals(messages.get(0).getText(), "Hello World!");  
  
tx.commit();
```

← ❶ 标准查询

- ❶ 一个 Hibernate 标准查询是表示查询的一种类型安全的编程方式，会其自动转译成 SQL。

本书中的大多数示例都未使用 `SessionFactory` 或 `Session` API。当不时出现仅在 Hibernate 中可用的特定功能时，我们将介绍给定一个标准 API 时如何对原生接口执行 `unwrap()` 操作。

2.4 本章小结

- 你已经完成了首个 JPA 项目。
- 你编写了一个持久化类及其带有注解的映射。
- 介绍了如何配置和引导一个持久化单元，以及如何创建 `EntityManagerFactory` 入口点。然后你调用了 `EntityManager` 与数据库进行交互，以存储和加载持久化域模型类的实例。
- 探讨了一些更高级的原生 Hibernate 引导程序和配置选项，以及等效的基础 Hibernate API、`SessionFactory` 和 `Session`。

第3章

域模型和元数据

3

本章内容简介:

- 探究 CaveatEmptor 示例应用程序
- 实现域模型
- 对象/关系映射元数据选项

第2章里的“Hello World”示例介绍了 Hibernate; 当然, 它对于理解具有复杂数据模型的现实环境应用程序的需求没有太多用处。在本书的后续内容中, 我们要使用一个更复杂的示例应用程序——CaveatEmptor, 一个在线拍卖系统——来阐释 Hibernate 和 Java 持久化(Caveat emptor(货物出门概不退换)意味着“让购买者自行注意”)。

JPA 2 中的主要特性

- JPA 持久化提供程序现在自动集成了一个 Bean 验证提供程序。当存储数据时, 该提供程序会自动验证持久化类上的约束。
- 添加了元模型 API。可以获取(但是不能变更)持久化单元中类的名称、属性和映射元数据。

我们将通过引入一个分层应用程序架构来开始对该应用程序的探讨。然后, 你要学习如何识别一个问题域的业务实体。你要创建这些实体及其属性的概念模型, 这称为域模型, 并且你要通过创建持久化类来在 Java 中实现它。我们要花一些时间来确切探究这些 Java 类看起来应该是什么样子、以及它们适用于典型分层应用程序架构中的什么地方。我们还要查看类的持久化功能以及这方面会如何影响设计和实现。我们要添加 Bean 验证, 它有助于自动验证域模型数据的完整性, 这不仅仅针对持久化信息, 而是针对所有业务逻辑。

之后我们要探究映射元数据选项——即告知 Hibernate 持久化类及其属性与数据库表和列关联起来的方法。这与直接在类的 Java 源代码中添加注解或者编写最终同已编译 Java 类一起部署的 XML 文档一样简单, Hibernate 在运行时会访问这些 Java 类。阅读完本章之后, 你将清楚如何设计复杂现实环境项目中域模型的持久化部分, 以及你会主要选用哪些映射元数据选项。我们这就开始探究该示例应用程序。

3.1 CaveatEmptor 示例应用程序

CaveatEmptor 示例是一个揭示 ORM 技术和 Hibernate 功能的在线拍卖应用程序，可以从 www.jpwh.org 下载该应用程序的源代码。本书中我们不会过多关注用户界面(它可能基于 Web 或者是一个富客户端)；而是要专注于数据访问代码。当关于数据访问代码的设计决策会引发必须制作用户界面的结果时，我们自然而然要考虑这两者。

为了理解 ORM 中涉及的设计问题，我们先假定 CaveatEmptor 应用程序还不存在，你要从头开始构建它。我们首先来查看其架构。

3.1.1 一个分层架构

对于所有重要的应用程序来说，通常根据关注点来组织类是合理的。持久化是一个关注点；其他关注点还包括呈现、工作流以及业务逻辑。一个典型的面向对象架构包括表示关注点的代码层。

横切关注点

还有所谓的横切关注点，一般它可能是由框架代码实现的。典型的横切关注点包括日志记录、授权以及事务分界。

一个分层架构会定义实现各种关注点的代码之间的接口，以允许在对一个关注点的实现方式进行变更时不会显著影响其他层中的代码。层次可以判定所发生的层间依赖性的类别。其规则如下：

- 层通信是自上而下的，一个层仅依赖其下一层的接口。
- 除了正好位于其下的层之外，每个层都不知道其他任何层。

不同的系统对关注点的分组也不同，因而它们会定义不同的层。典型的经过验证的高级别应用程序架构会使用三层：分别用于呈现、业务逻辑和持久化，如图 3-1 所示。

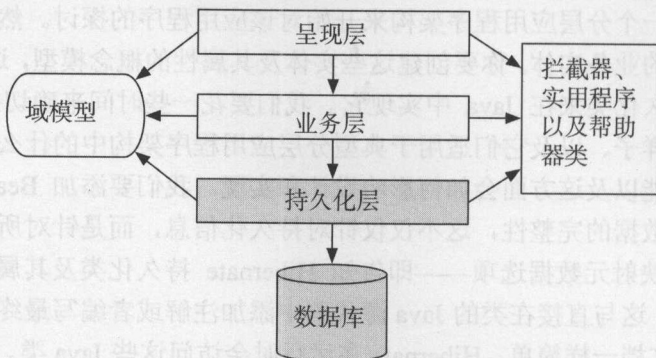


图 3-1 持久化层是分层架构中的基础

- 呈现层——用户界面逻辑位于最上层。负责呈现以及页面和界面导航控制的代码位于呈现层中。用户界面代码可以直接访问共享域模型的业务实体并将其呈现在界面上，以及访问执行操作的控件。在某些架构中，业务实体实例可能不能直接

由用户界面代码访问：例如，如果呈现层不像系统的其余部分那样运行在相同的计算机上就会出现这种情况。在这种情形下，呈现层可能会需要其专用的数据传输模型，以便只表示该域模型的一个可传输子集。

- 业务层——紧接着的下一层，其确切形式在各应用程序之间有很大的不同。通常认为业务层负责实现所有业务规则或系统需求，它们会被用户理解为问题域的一部分。这一层通常包括一些类型的控制组件——清楚何时调用哪个业务规则的代码。在某些系统中，这个层有其自己的业务域实体的内部表示。或者，它依赖于一个域模型实现，与该应用程序的其他层共享。
- 持久化层——持久化层是一组类和组件，它们负责将数据存储到一个或多个数据存储中并从中检索这些数据。这一层需要业务域实体的一个模型，你要为其保存持久化状态。持久化层就是大多数 JPA 和 Hibernate 发挥作用的地方。
- 数据库——数据库通常位于外部，由多个应用程序共享。它是系统状态的真实持久化表示。如果使用了一个 SQL 数据库，则该数据库会包括一个架构，并且可能还有用于执行与数据相关的业务逻辑的存储过程。
- 帮助器和实用类——每个应用程序都有一组基础帮助类或者实用类，它们用于应用程序的每一层(比如用于错误处理的 Exception 类)。这些共享的基础元素不构成一个层，这是因为它们并不遵循分层架构中层间依赖的规则。

现在你已经有一个高层架构，可以专注于业务问题了。

3.1.2 分析业务域

在这一阶段，你要寻求域专家的帮助，分析你的软件系统需要解决的业务问题，识别出相关主要实体及其交互。域模型的分析 and 设计背后激励人心的目标就是，为应用程序的目标捕获业务信息的本质。

实体通常是系统用户所理解的概念：支付、客户、订单、商品项、竞拍价等。一些实体可能是用户考虑较不具体的事物的抽象，比如定价算法，但即便是这些实体，通常也是用户能够理解的。可以在业务的概念视图找到所有这些实体，该概念视图有时候也称为业务模型。

根据此业务模型，工程师和面向对象的软件架构师会创建一个面向对象的模型，仍旧是概念阶段(没有 Java 代码)。这个模型可能就像仅仅存在于开发人员脑海中的思维图形那么简单，或者可能像一张 UML 类图那样制作精细。图 3-2 显示了在 UML 中表述的一个简单模型。

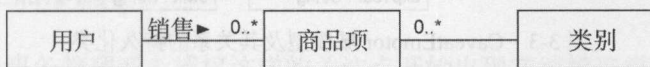


图 3-2 典型拍卖网站的一个类图

这个模型包含你一定会在任何典型电子商务系统中找到的实体：类别、商品项以及用户。该问题域的这一模型表示了所有的实体及其关系(可能还有其属性)。我们将此类来自问题域实体的面向对象模型称为域模型，它仅包含用户关注的那些实体。域模型是现实环境的一个抽象视图。

相较面向对象的模型而言，工程师和架构师可以使用一个数据模型(可能用一个实体-关系图来表示)开始应用程序的设计。我们通常说，就持久化而言，这两者之间没什么区别；它们只是不同的起始点。最终，你所使用的建模语言是次要的；我们最关心的是业务实体的结构和关系。我们关注必须应用以确保数据完整性的规则(例如关系的多样性)以及用于操作数据的代码过程。

在下一节中，我们要完成 CaveatEmptor 问题域的分析。所产生的域模型将是本书的中心主题。

3.1.3 CaveatEmptor 域模型

CaveatEmptor 网站可以拍卖许多不同种类的商品项，从电子设备到机票。拍卖过程依据英国拍卖策略进行：用户连续在一件商品上出价，直到该商品的竞价时间结束，最高出价者购得该商品。

在所有的商店中，货物都是按照类别进行分类的，并且类似的货物分组到一个分区并上架。拍卖目录需要商品类别的某种层次结构，以便购买者可以浏览这些类别或者根据类别和商品属性随意搜索。商品清单会显示在类别浏览器和搜索结果屏幕上。从清单中选择一件商品会为购买者呈现商品详情的视图，其中可能附有该商品的图片。

一次拍卖由一系列出价构成，其中有一个是竞拍成功的出价。用户详情包括姓名、地址和账单信息。

图 3-3 显示了此分析的结果，即该域模型的高层次概览。我们简要探讨一下这个模型的一些有趣特性。

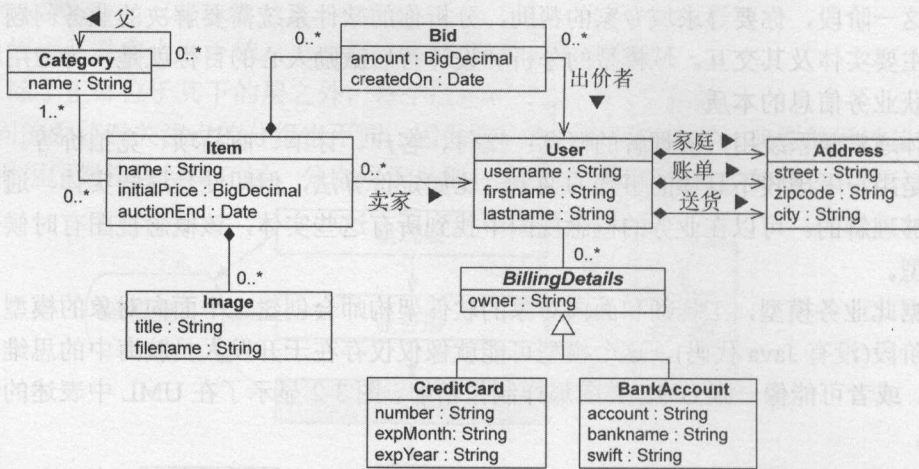


图 3-3 CaveatEmptor 域模型及其关系的持久化类

每件商品只能拍卖一次，所以你不需要将 Item 与其他所有拍卖实体区别开来。相反，你有一个名为 Item 的拍卖商品实体。因此，Bid 直接与 Item 相关联。你要将 User 的 Address 信息建模成一个独立的类。一个 User 可能具有三个地址：家庭、账单和送货地址。必须允许用户拥有多个 BillingDetails。一个抽象类的各个子类表示了各种结算策略(允许未来进行扩展)。

该应用程序可将一个 Category 嵌套在另一个 Category 中，依次类推。从 Category 实体到其自身的递归关联表示了这一关系。注意，单个 Category 可能具有多个子类别，但最多只有一个父类别。每个 Item 至少属于一个 Category。

这一表示并非完整的域模型，而仅仅是需要持久化功能的类。你会要存储和加载 Category、Item、User 等实例。我们对这一高层次概览进行了一些简化；在更复杂的示例有所需要时，我们可以随后引入附加类或者对它们稍作修改。

当然，域模型中的实体应该封装状态和行为。例如，User 实体应该定义客户的姓名和地址，以及计算(为此特定客户)进行商品运输的成本所需的逻辑。

该域模型中可能有其他只具有瞬态运行时实例的类。思考一下封装了最高出价者赢得拍卖这一事实的一个 WinningBidStrategy 类。在检查一次拍卖的状态时，这个类可能会被业务层(控制器)调用。有时候，你可能必须弄明白销售商品的税如何计算，或者系统如何审核通过一个新用户账户。我们不是认为这样的业务规则或域模型行为是不重要的；而是因为这个问题与持久化问题基本上是正交的。

现在你已经有一个带有域模型的(基础)应用程序设计，下一步就是在 Java 中实现它。

不使用域模型的 ORM

使用完整 ORM 的对象持久化最适合基于富域模型的应用程序。如果你的应用程序不实现完整业务规则或实体之间的复杂交互(或者只有很少的实体)，那么你可能并不需要域模型。许多简单以及一些不那么简单的问题都可完美适用于面向表的解决方案，其中应用程序是围绕数据库数据模型来设计的，而不是围绕面向对象的域模型来设计，并且通常还具有在数据库(存储过程)中执行的逻辑。另一个要考虑的方面是学习曲线：一旦你精通了 Hibernate，就会将其用于所有的应用程序，即使是用作简单 SQL 查询生成器以及结果映射器。如果才开始学习 ORM，那么一个简单的用例可能并不能证明你所投入的时间和成本是物有所值的。

3.2 实现域模型

你要从一个所有实现都必须处理的问题开始：关注点分离。域模型实现通常是一个集中组织的组件；每当实现新的应用程序功能时都要大量重用它。基于此原因，你应该准备好尽力确保除了业务方面的关注点之外，其他关注点不会渗入到域模型的实现之中。

3.2.1 处理关注点渗漏

在像持久化、事务管理或者授权这样的关注点开始出现在域模型类中时，就出现了关注点渗漏的一个例子。域模型实现是代码如此重要的部分，它不应该依赖互不相关的 Java API。例如，域模型中的代码不应该执行 JNDI 查找或者通过 JDBC API 调用数据库，直接访问或者通过一个中间抽象访问都不行。这样就能够在几乎任何位置重用域模型类：

- 在呈现视图时，呈现层可以访问域模型实体的实例和属性。

- 业务层中的控制器组件还可以访问域模型实体的状态，并且调用实体方法来执行业务逻辑。
- 持久化层可以从数据库加载域模型实体的实例并将其存储到数据库，以保存其状态。

最重要的是，避免关注点渗漏使得对域模型进行单元测试变得容易，而不需要特定的运行时环境或容器，也不需要模拟任何服务依赖。可以编写验证域模型类正确行为的单元测试，无需任何特殊的测试工具(我们不是在谈论与“从数据库加载”以及“存储到数据库”有关的测试内容，而是讨论“计算运输成本和税费”的行为)。

Java EE 标准使用元数据解决了关注点渗漏的问题，这些元数据作为注解存在于你的代码中，或者作为 XML 描述符外部化。这一方式使得运行时容器可以实现一些预定义的横切关注点——安全、并发、持久化、事务以及远程问题——以通过拦截对应用程序组件的调用这一通用方式。

Hibernate 并非一种 Java EE 运行时环境，也不是应用程序服务器。它只是 Java EE 保护伞(JPA)之下的一种规范的实现，并且只是用于持久化这个关注点的一个解决方案。

JPA 将实体类定义为主要的编程构件。这一编程模型使得透明持久化成为可能，并且像 Hibernate 这样的 JPA 提供程序还提供了自动持久化。

3.2.2 透明及自动持久化

我们使用透明一词，意味着域模型的持久化类和持久化层之间的关注点完全分离。持久化类不清楚——且不依赖——持久化机制。我们使用自动一词来谈及一种持久化解决方案(你注解的域、层和机制)，它能让你免于处理像编写大多数 SQL 语句和处理 JDBC API 这样的低层次机械化细节。

例如，CaveatEmptor 域模型的 Item 类不应该有对于任何 Java 持久化或 Hibernate API 的任何运行时依赖。此外：

- JPA 不需要任何特殊超类或接口被持久化类继承或实现，也不需要任何用于实现属性和关联的特殊类(当然，使用这两种技术的选项一直是存在的)。
- 可以在持久化的上下文之外重用持久化类，比如在单元测试或持久化层中。可以使用常规的 Java new 操作符在任何运行时环境中创建实例，以保持可测试性以及可重用性。
- 在具有透明持久化的系统中，实体的实例并不清楚底层的数据存储；它们甚至不需要清楚其正在持久化或检索。JPA 将持久化关注点向一个通用的持久化管理器 API 外部化。
- 因此，你的大多数代码(当然也包括复杂业务逻辑)本身不必关注单个执行线程中域模型实体实例的当前状态。

我们认为透明是必需的，因为它会让应用程序更易于构建和维护。透明持久化应该是所有 ORM 解决方案的一个主要目标。显然，没有自动持久化解决方案是完全透明的：每一个自动持久化层，包括 JPA 和 Hibernate，都会把一些需求强加在持久化类上。例如，JPA 要求集合的属性输入到一个接口，比如 `java.util.Set` 或者 `java.util.List`，而不是一个实际的

实现, 比如 `java.util.HashSet`(这无论如何都是一种良好的做法)。或者, 一个 JPA 实体类必须具有一个特殊属性, 它称为数据库标识符(这样也受到更少的限制, 但通常很便利)。

现在你知道为何持久化机制应该对你如何实现域模型具有最小的影响, 以及为何需要透明和自动持久化。我们推荐的达成此目的的编程模型是 POJO。

POJO

POJO 是简单传统 Java 对象(Plain Old Java Objects)的简写, Martin Fowler、Rebecca Parsons 和 Josh Mackenzie 于 2000 年创造了这个词。

大概 10 年前, 许多开发人员开始谈论 POJO, 它是一种回归基本的方法, 在本质上复兴了 JavaBeans 这一用于 UI 开发的组件模型, 并且这些开发人员将 POJO 重新应用到了系统的其他层。EJB 和 JPA 规范的一些修订版为我们带来了新的轻量级实体, 并且可适当地将它们称为可持久化 JavaBeans。Java 工程师通常使用所有这些术语作为相同基本设计方法的同义词。

你不应该过于关注我们在本书中使用了哪些术语; 最终的目标是尽可能透明地将持久化方面应用到 Java 类。如果遵循一些简单做法的话, 那么几乎任何 Java 类都可以持久化。我们来看看在代码中如何实现。

3.2.3 编写可持久化类

处理细粒度且丰富的域模型是一个重要的 Hibernate 目标。这是我们使用 POJO 的原因。一般来说, 使用细粒度的对象意味着类比表要多。

一个可持久化的简单传统 Java 类要声明表示状态的属性以及定义行为的业务方法。一些属性表示与其他可持久化类的关联。

代码清单 3.1 显示了域模型的用户实体的一个 POJO 实现。我们浏览一下该代码。

代码清单3.1 User类的POJO实现

路径: `/model/src/main/java/org/jpwh/model/simple/User.java`

```
public class User implements Serializable {  
  
    protected String username;  
  
    public User() {  
    }  
  
    public String getUsername() {  
        return username;  
    }  
  
    public void setUsername(String username) {  
        this.username = username;  
    }  
  
    public BigDecimal calcShippingCosts(Address fromLocation) {  
        // Empty implementation of business method  
        return null;  
    }  
}
```

```
}  
// ...  
}
```

JPA 不需要持久化类实现 `java.io.Serializable`。但是当实例存储在一个 `HttpSession` 中或者使用 RMI 通过值传递时，就需要序列化。尽管这可能不会出现在你的应用程序中，但该类将在无需任何额外工作的情况下被序列化，并且声明它不会有任何弊端(我们打算在每个示例中声明它，假定你知道它何时是必需的)。

该类可以是抽象的，并且如有必要，它可以扩展一个非持久化类或者实现一个接口。它必须是一个顶层类，不嵌套在另一个类中。可持久化类及其所有的方法都不能是最终的(JPA 规范的一个要求)。

不同于不需要任何特定构造函数的 `JavaBeans` 规范，`Hibernate`(和 JPA)需要让每个持久化类具有不带有参数的构造函数。或者，你可能完全不编写构造函数；这样 `Hibernate` 将使用 `Java` 的默认构造函数。`Hibernate` 会使用这样一个无参构造函数上的 `Java` 反射 API 来调用类，以便创建实例。该构造函数可以不是公共的，但如果为了优化性能，`Hibernate` 要使用运行时生成的代理，则它必须至少是包可见的。还有，考虑一下其他规范的要求：`EJB` 标准要求在会话 `bean` 构造函数上实现公共可见性，就像 `JavaServer Faces(JSF)` 规范需要其托管 `bean` 一样。当想要一个公共构造函数来创建一个“空”状态时，会出现其他情形：例如，按示例构建查询。

`POJO` 的属性实现了业务实体的属性——例如 `User` 的 `username`。你通常要将属性作为私有或受保护成员字段，与公共或受保护属性访问器方法一起实现：对于每个字段，要有一个检索其值的方法和一个设置其值的方法。这些方法分别称为获取方法和设置方法。代码清单 3.1 中的 `POJO` 示例为 `username` 属性声明了获取方法和设置方法。

`JavaBean` 规范为访问器方法的命名定义了指导原则；这使得像 `Hibernate` 这样的通用工具可以轻易发现和操作属性值。获取方法的名称以 `get` 开头，后面跟着该属性的名称(首字母大写)；设置方法的名称以 `set` 开头，类似地，其后面跟着该属性的名称。用于 `Boolean` 属性的获取方法可以用 `is` 替代 `get` 作为开头。

`Hibernate` 不需要访问器方法。可以选择应该如何持久化你的持久化类实例的状态。`Hibernate` 要么会直接访问字段，要么会调用访问器方法。这些考虑事项对类设计的干扰不会太大。可以让一些访问器方法变成非公共的，或者完全移除它们——之后配置 `Hibernate` 来依赖这些属性的字段访问。

属性字段和访问器方法应该是私有的、受保护的或包可见的？

通常，你希望阻止对类的内部状态进行直接访问，因而你不会让属性字段变成公共的。如果让字段或方法成为私有，则实际上是在声明不应该访问它们；只有在你允许的情况下(或者像 `Hibernate` 这样的服务)才能访问它们。这是一个明确的声明。某些人访问你的“私有”内部对象通常有合理的原因——通常是修补你的其中一个 `bug`——而如果在紧急情况下他们必须借助反射访问的话，那么这样做只会让他们感到愤怒。相反，你可能会假设或者清楚在你之后的工程师已经访问了你的代码，并且知道他们在做什么。

如此，受保护的可见性就是一个更合理的默认选项。你禁止了直接的公共访问，表明

这个特定成员细节是内部信息，但在需要的情况下允许由子类访问。你信任那些创建该子类的工程师。包可见性是无理的：你强制某些人在相同的包中创建代码来访问成员字段和方法；这是毫无意义的额外工作。最重要的是，这些可见性的推荐做法是与环境相关的，而无需安全性策略和运行时 SecurityManager。如果必须保持你的内部代码为私有，那就让它私有化吧。

尽管寻常的访问器方法是共用的，但我们喜欢使用 JavaBeans 风格访问器方法的一个原因是，它们提供了封装性：可以修改一个属性的隐藏内部实现，而无须对公共接口进行任何变更。如果配置 Hibernate 通过方法访问属性，则是将类的内部数据结构(实例变量)从数据库设计中抽象出来。

例如，如果你的数据库将一个用户的姓名，存储为单个 NAME 列，而 User 类具有 firstname 和 lastname 字段，则可以将以下持久化 name 属性添加到该类。如代码清单 3.2 所示。

代码清单3.2 在访问器方法中具有逻辑的User类的POJO实现

```
public class User {  
  
    protected String firstname;  
    protected String lastname;  
  
    public String getName() {  
        return firstname + ' ' + lastname;  
    }  
  
    public void setName(String name) {  
        StringTokenizer t = new StringTokenizer(name);  
        firstname = t.nextToken();  
        lastname = t.nextToken();  
    }  
}
```

后面会看到，持久化服务中的一个自定义类型转换器是处理许多此类情形的一种更好的方式，它有助于使用若干选项。

另一个要考虑的问题是脏检查。Hibernate 会自动检测状态变更以便保持与数据库中更新后状态的同步。从获取方法返回一个不同的实例通常比返回由 Hibernate 传递给设置方法的实例要安全。Hibernate 会根据值——而不是根据对象标识——来比较它们，以确定该属性的持久化状态是否需要更新。例如，以下获取方法不会导致执行不必要的 SQL UPDATE:

```
public String getFirstname() {  
    return new String(firstname);  
}
```

←—— 这是可行的。

这样做有一个重要的特例：集合是通过标识来对比的！对于映射为一个持久化集合的属性，你应该从获取方法中准确返回与 Hibernate 传递给设置方法的实例相同的集合实例。如果不这样做，则 Hibernate 将更新数据库，即便不需要更新也会如此，每次保留在内存中的状态都会与数据库同步。你通常应该避免在访问器方法中使用此类代码：

```
protected String[] names = new String[0];

public void setNames(List<String> names) {
    this.names = names.toArray(new String[names.size()]);
}

public List<String> getNames() {
    return Arrays.asList(names);
}
```

如果 Hibernate 要访问这
些方法，则不要这么做！

当然，如果 Hibernate 直接访问 `names` 字段，忽略你的获取和设置方法，这样做就不会有问题。

当访问器方法抛出异常时，Hibernate 如何处理它们呢？当加载和存储实例且抛出一个（不受检测的）`RuntimeException` 时，如果 Hibernate 使用访问器方法，则当前事务会回滚，且该异常需要在称为 Java 持久化(或 Hibernate 原生)API 的代码中进行处理。如果抛出一个受检测的应用程序异常，则 Hibernate 会将该异常包装到一个 `RuntimeException` 中。

代码清单 3.2 中的示例还定义了一个计算将商品运输到某个特定用户的成本的业务方法(我们省略了这个方法的实现)。

接下来，我们要专注于实体间的关系和持久化类之间的关联。

3.2.4 实现 POJO 关联

现在你将看到如何进行关联以及如何创建对象之间不同的关系类型：一对多、多对一以及双向关系。我们将查看创建这些关联所需的脚手架代码，如何简化关系管理，以及如何加强这些关系的完整性。

你要创建属性来表达类之间的关联，并且(通常)要调用访问器方法在运行时进行实例间的导航。我们思考一下由 `Item` 和 `Bid` 持久化类定义的关联，如图 3-4 所示。

就像我们所有的 UML 类图一样，这里省略了与关联有关的属性 `Item#bids` 和 `Bid#item`。这些属性和操作其值的方法称为脚手架代码。下面是用于 `Bid` 类的脚手架代码的示例：

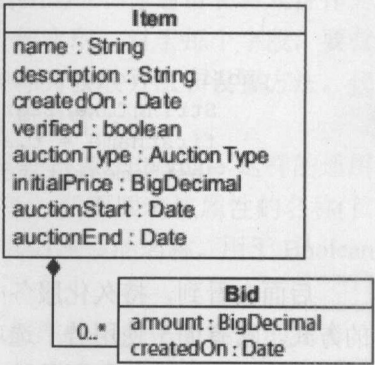


图 3-4 Item 和 Bid 类之间的关联

路径: /model/src/main/java/org/jpwh/model/simple/Bid.java

```
public class Bid {

    protected Item item;

    public Item getItem() {
        return item;
    }

    public void setItem(Item item) {
        this.item = item;
    }
}
```

`item` 属性允许从一个 `Bid` 导航到相关 `Item`。这是具有多对一的多性的一个关联；用户可以为每件商品出价多次。下面是 `Item` 类的脚手架代码：

路径：/model/src/main/java/org/jpwh/model/simple/Item.java

```
public class Item {
    protected Set<Bid> bids = new HashSet<Bid>();

    public Set<Bid>getBids() {
        return bids;
    }

    public void setBids(Set<Bid> bids) {
        this.bids = bids;
    }
}
```

两个类之间的这种关联可以采用双向导航：从这个角度看，多对一就是一种一对多的多样性(同样，一件商品可以有多次出价)。用于 `bids` 属性的脚手架代码使用了一个集合接口类型 `java.util.Set`。JPA 需要用于集合类型属性的接口，因而你必须使用 `java.util.Set`、`java.util.List` 或者 `java.util.Collection`，而不是像 `HashSet` 这样的接口。无论如何，相对于具体实现，为集合接口编程都是良好的做法，因此这一限制应该不会干扰到你。

你选择了一个 `Set` 并将该字段初始化为一个新的 `HashSet`，因为应用程序不允许重复出价。这是不错的做法，因为当有人在未出价的情况下访问新 `Item` 的属性时，这样做就可以避免出现任何 `NullPointerExceptions`。要在任何映射的集合的属性上设置一个非空值，也需要用到 JPA 提供程序：例如，在从数据库加载不带有出价的 `Item` 时就需要用到(不必非得使用 `HashSet`；具体实现取决于提供程序。`Hibernate` 具有自己的带有附加功能的集合实现——例如脏检查)。

一件商品上的出价不应该存储到列表中吗？

第一反应通常是按照用户输入的顺序来保存元素，因为这可能也是后续要显示它们的顺序。当然，在一个拍卖应用程序中，必须有一些定义好的顺序，让用户按照这些顺序看到一件商品的出价——例如，最高出价排第一或者最新出价排最后。你甚至可以在用户界面代码中使用 `java.util.List` 来排序和显示商品的出价。这并不意味着这一显示顺序应该是持久的；数据完整性不会被出价显示的顺序所影响。需要存储每次出价的金额，以便可以找出最高出价，并且需要存储创建每次出价时的时间戳，以便找出最新出价。在把握时，要保持你系统的灵活性，在从数据存储检索数据(在查询中)并且/或者显示给用户(在 Java 代码中)时对数据排序，而不是在存储数据时排序。

就像用于基本属性的访问器方法一样，只有在作为应用程序逻辑用来创建两个实例之间链接的持久化类的外部接口一部分时，用于关联的访问器方法才需要声明为 `public` 类型。现在我们将专注于这一问题，因为在 Java 代码中管理 `Item` 和 `Bid` 之间的链接远比在具有

声明式外键约束的 SQL 数据库中复杂。根据我们的经验,工程师通常不会意识到具有双向引用(指针)的网络对象模型所产生的这一复杂性。让我们循序渐进地探讨这个问题。

用于链接 Bid 和 Item 的基本程序看起来如下所示:

```
anItem.getBids().add(aBid);  
aBid.setItem(anItem);
```

无论何时创建这一双向链接,都需要两个操作:

- 必须将该 Bid 添加到 Item 的 bids 集合。
- 必须设置该 Bid 的 item 属性。

JPA 不管理持久化关联。如果希望操作一个关联,则必须编写与没有 Hibernate 的情况下所编写的完全相同的代码。如果一个关联是双向的,则必须考虑该关系的两面。如果曾经遇到过理解 JPA 中关联行为的问题,则可以问你自己,“没有 Hibernate 我会怎么做?” Hibernate 不会改变常规的 Java 语义。

我们建议你添加分组这些操作的便利方法,从而允许重用和帮助确保正确性,并最终确保数据的完整性(Bid 必须具有对 Item 的引用)。代码清单 3.3 显示了 Item 类中这样的便利方法。

代码清单3.3 一个便利方法会简化关系管理

路径: /model/src/main/java/org/jpwh/model/simple/Item.java

```
public void addBid(Bid bid) {  
    if (bid == null) ←—— 具有保护意识  
        throw new NullPointerException("Can't add null Bid");  
    if (bid.getItem() != null)  
        throw new IllegalStateException("Bid is already assigned to an  
        Item");  
    getBids().add(bid);  
    bid.setItem(this);  
}
```

addBid()方法不仅会减少处理 Item 和 Bid 实例时的代码量,还会强制实施关联的基数。你避免了从省略这两个所需操作之一所产生的错误。如果可能的话,应该总是提供此类用于关联操作的分组。如果将其与 SQL 数据库中外键的关系模型作比较,就会轻易看出网络和指针模型是如何使一个简单操作变得复杂的:相对于一个声明式约束,需要过程代码来确保数据完整性。

由于你希望 addBid()是用于商品出价的唯一外部可见的修改器方法(除此之外,可能还有一个 removeBid()方法),因此可以让 Item#setBids()方法变为私有,或者删除它并配置 Hibernate 来直接访问用于持久化的字段。思考一下出于相同原因让 Bid#setItem()方法变为包可见的情况。

Item#getBids()获取方法仍旧会返回一个可修改集合,因此客户端可以使用它来进行不会在另一端反映出来的变更。直接添加到该集合的出价不会引用一件商品——就你的数据库约束而言,这是一种不一致状态。为避免这种情形,在从获取方法返回内部集合之前,可使用 Collections.unmodifiableCollection(c)和 Collections.unmodifiableSet(s)包装该集合。之

后如果客户端尝试修改该集合，则会产生一个异常；这样就可以强制每次修改都要经过该关系管理方法的审查，从而确保完整性。注意，在这种情况下，必须为字段访问配置 Hibernate，因为之后由获取方法返回的集合与指派给设置方法的集合并不相同。

一个备选策略是不可变实例。例如，可以通过在 Bid 的构造函数中要求一个 Item 参数来强制实施完整性，如代码清单 3.4 所示。

代码清单3.4 使用一个构造函数强制实施关系完整性
路径: /model/src/main/java/org/jpwh/model/simple/Bid.java

```
public class Bid {  
    protected Item item;  
  
    public Bid(Item item) {  
        this.item = item;  
        item.getBids().add(this); ← 双向的  
    }  
    public Item getItem() {  
        return item;  
    }  
}
```

在这个构造函数中设置了 item 字段；不应该出现对该字段值的进一步修改。还更新了双向关系中“另一”端上的集合。此处没有 Bid#setItem()方法，并且你可能不应该公开一个公共的 Item#setBids()方法。

这一方法有几个问题。首先，Hibernate 不能调用这个构造函数。需要为 Hibernate 添加一个无参构造函数，并且它至少需要是包可见的。此外，因为没有 setItem()方法，所以必须配置 Hibernate 以直接访问 item 字段。这意味着该字段不能是 final 类型，因此不能确保这个类是不可变的。

在本书的示例中，我们有时会编写脚手架方法，比如之前介绍的 Item#addBid()；或者我们可以为所需的值使用额外的构造函数。围绕持久化关联属性和/或字段要包装多少便利方法和层取决于你自己，但我们建议保持一致，并且将相同的策略应用到所有的域模型类。为了实现可读性，我们不会总是在未来的代码示例中显示便利方法、特殊构造函数以及其他诸如此类的脚手架代码，并且假定你会根据自己的偏好和需要来添加它们。

现在你已经看到了域模型类、如何表示其属性以及它们之间的关系。接下来，我们要提高抽象的级别，将元数据添加到域模型实现，并且声明像验证和持久化规则这样的方面。

3.3 域模型元数据

元数据是与数据有关的数据，所以域模型元数据是与你的域模型有关的信息。例如，当使用 Java 反射 API 来发现域模型的类名称或者其属性的名称时，就是在访问域模型元数据。

ORM 工具也需要元数据来指定类和表、属性和列、关联和外键、Java 类型和 SQL 类型等之间的映射。这一对象/关系映射元数据会管治不同类型系统与面向对象和 SQL 系统中关系表示之间的转换。JPA 具有一个元数据 API，可以调用它来获得与域模型持久化方面有关的详情，比如持久化实体和属性的名称。首先，作为工程师，你的任务是创建并维护这一信息。

JPA 标准化了两个元数据选项：Java 代码中的注解以及外部化的 XML 描述符文件。Hibernate 具有用于原生功能的一些扩展，也是作为注解和/或 XML 描述符来提供的。通常我们要么选择注解，要么选择 XML 文件作为映射元数据的主要来源。在阅读完本节后，你将获得相关的背景信息，以便对自己的项目做出明智的决定。

我们还将探讨 Bean 验证(JSR303)以及它如何为你的域模型(或任何其他)类提供声明式验证。这一规范的参考实现是 Hibernate 验证器项目。如今，大多数工程师都选用 Java 注解作为声明元数据的主要机制。

3.3.1 基于注解的元数据

注解的一大优势在于，将元数据置于它所描述的信息旁边，而不是将其物理分隔到另一个文件中。下面是一个示例：

路径：/model/src/main/java/org/jpwh/model/simple/Item.java

```
import javax.persistence.Entity;
```

```
@Entity
```

```
public class Item {
```

```
}
```

可以在 `javax.persistence` 包中找到标准的 JPA 映射注解。这个示例使用 `@javax.persistence.Entity` 注解将 `Item` 类声明为一个持久化实体。现在其所有属性都会使用一个默认策略自动持久化。这意味着可以加载和存储 `Item` 的实例，并且该类的所有属性都是托管状态的一部分。

如果阅读过第 2 章，你大概会注意到遗漏了所需的 `@Id` 注解以及标识符属性。如果希望尝试练习该 `Item` 实例，则必须添加一个标识符属性。我们将在 4.2 节中再次探讨标识符属性。

注解是类型安全的，并且编译的类文件中包括了 JPA 元数据。然后当应用程序启动时，Hibernate 会使用 Java 反射读取这些类和元数据。IDE 还可以轻易验证和强调注解——毕竟，它们是常规的 Java 类型。如果重构你的代码，就始终要重命名、删除或者移动类和属性。大多数开发工具和编辑器都不能重构 XML 元素和属性值，但注解是 Java 语言的一部分，并且包含在所有重构操作中。

现在我的类依赖 JPA 吗？

是的，但仅在编译时依赖。在编译你的域模型类的来源时，你的类路径上需要 JPA 库。当创建类的一个实例时，类路径上不需要 Java 持久化 API：例如，在一个桌面客户端应用

程序中，不会执行任何 JPA 代码。只有在运行时通过反射访问注解时(正如 Hibernate 读取你的元数据时在内部所做的那样)，类路径上才需要该包。

当标准化 Java 持久化注解不够用时，JPA 提供程序可以提供额外注解。

1. 使用供应商扩展

即使你用 `javax.persistence` 包中 JPA 兼容的注解映射了大多数的应用程序模型，也不得不在某些时候使用供应商扩展。例如，你期望在高质量持久化软件中可用的一些性能调优选项只能作为特定于 Hibernate 的注解提供。这就是 JPA 提供程序的竞争方式，这样就无法避免使用其他包的注解——这也是你选择 Hibernate 的一个原因。

下面又是一个只使用 Hibernate 映射选项的 Item 实体源代码：

```
import javax.persistence.Entity;

@Entity
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
)
public class Item {
}
```

我们更喜欢使用完整的 `org.hibernate.annotations` 包名作为 Hibernate 注解的前缀。建议遵循这一良好做法，因为你会轻易发现用于这个类的哪些元数据来自 JPA 规范，哪些又是特定于供应商的。你还可以轻松地使用“`org.hibernate.annotations`”搜索你的源代码，并在单个搜索结果中得到应用程序中所有非标准注解的完整总览。

如果要切换 Java 持久化提供程序，那么只需要替换特定于供应商的扩展即可，可以预期大多数成熟的 JPA 实现都有一个类似的功能集可用。当然，我们希望你永远不必这样做，并在实践中它不会频繁出现——只要做好准备即可。

类上的注解仅涵盖了适用于特定类的元数据。你通常需要较高层次的元数据，以用于整个包甚或整个应用程序。

2. 全局注解元数据

`@Entity` 注解映射了一个特定类。JPA 和 Hibernate 还具有用于全局元数据的注解。例如，`@NamedQuery` 具有全局作用域；你不应将其应用于一个特定类。那么，你应该将这个注解放在哪里呢？

尽管在一个类(确实是位于顶层的任何类)的源文件中放置这样的全局注解是可行的，但我们更愿意在一个单独文件中保存全局元数据。包级别的注解是一个很好的选择；它们位于特定包目录中称为 `package-info.java` 的文件内。在代码清单 3.5 中，可以看到全局命名查询声明的一个示例。

代码清单3.5 package-info.java文件中的全局元数据

路径: /model/src/main/java/org/jpwh/model/querying/package-info.java

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByName",
        query = "select i from Item i order by i.name asc"
    )
    ,
    @org.hibernate.annotations.NamedQuery(
        name = "findItemBuyNowPriceGreaterThan",
        query = "select i from Item i where i.buyNowPrice > :price",
        timeout = 60,
        comment = "Custom SQL comment"
    )
})
package org.jpwh.model.querying;
```

秒数!

除非你之前已经使用过包级别的注解,否则这个文件使用包的语法以及位于底部的导入声明对于你来说可能是全新的内容。

上面的代码示例只包含来自 Hibernate 包的注解且不包含 Java 持久化注解是有原因的。我们省略了标准的 JPA `@org.javax.persistence.NamedQuery` 注解,并且使用 Hibernate 作为替代。JPA 注解不具有包适用性——我们不知道原因所在。实际上, JPA 不允许在 `package-info.java` 文件中使用注解。原生的 Hibernate 注解提供了相同的(有时候甚至更多的)功能,所以这应该不会是太大的问题。如果不希望使用 Hibernate 注解,则必须在所有类的顶部放置 JPA 注解(可以使用另外的空 `MyNamedQueries` 类作为域模型的一部分),或者使用一个 XML 文件,在本节稍后你会介绍。

注解将是本书全篇内容中用于 ORM 元数据的主要工具,关于这个主题有很多内容需要学习。在我们介绍一些可供选择的使用 XML 文件的映射方式之前,通过一些简单注解来加入验证规则以改进该域模型类。

3.3.2 应用 Bean 验证规则

大多数应用程序都包含大量的数据完整性检查。你已经看到了违背一个最简单的数据完整性约束时会发生什么:当期望一个值是可用时,你会得到一个 `NullPointerException`。其他的例子是,一个不应该为空的字符串值属性(记住,空字符串不是 `null`)、一个必须匹配特定正则表达式模式的字符串以及一个必须在某特定范围内的数值或者日期值。

这些业务规则会影响一个应用程序的每一层:用户界面代码必须显示详细的和本地化的错误消息。在将从客户端接收到的输入值传递给数据存储之前,业务和持久化层必须检查这些值。SQL 数据库必须是最后的验证器,最终确保持久数据的完整性。

Bean 验证背后的理念在于,声明像“这个属性不能为 `null`”或者“这个数值必须在指定范围内”这样的规则要比反复编写 `if-then-else` 程序更容易,并且较为不易出错。此外,在应用程序的中心组件(即域模型实现)上声明这些规则使得在系统的每个层中进行完整性

检查成为可能。然后这些规则就可用于呈现和持久化层。并且如果考虑一下数据完整性不仅会影响你的 Java 应用程序代码，还会影响你的 SQL 数据库模式——也就是一组完整性规则——你可能就会考虑将 Bean 验证约束用作额外的 ORM 元数据了。

看看代码清单 3.6 所示的扩展 Item 域模型类。

代码清单3.6 在Item实体字段上应用验证约束

路径: /model/src/main/java/org/jpwh/model/simple/Item.java

```
import javax.validation.constraints.Future;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
public class Item {

    @NotNull
    @Size(
        min = 2,
        max = 255,
        message = "Name is required, maximum 255 characters."
    )
    protected String name;
    @Future
    protected Date auctionEnd;
}
```

你新添加了两个属性：一件商品的 `name` 以及 `auctionEnd` 日期，即一次拍卖结束的日期。这两个属性都是用于额外约束的典型候选对象：你希望确保名称总是显示，并且可被人理解(单字符商品名称没有太多意义)，但它不应该过长——你的 SQL 数据库在可变长度字符串最多为 255 个字符时最为高效，并且你的用户界面在可见标签空间上也有一些约束。一次拍卖的结束时间显然应该是未来的某个时间。如果不提供一条错误消息，则会使用默认消息。消息对于国际化的外部属性文件来说可能是关键的内容。

如果注解字段的话，验证引擎将直接访问这些字段。如果选择通过访问器方法调用，则要注解带有验证约束的获取方法而非设置方法。之后约束就会成为类的 API 的一部分，并且包含在其 Javadoc 中，让域模型的实现更易于理解。注意这是独立于通过 JPA 提供程序访问权的；也就是说，Hibernate 验证器可以调用访问器方法，而 Hibernate ORM 可以直接调用字段。

Bean 验证不限于内置的注解；可以创建自己的约束和注解。有了自定义约束，你甚至可以使用类级别的注解，并且在类的实例上同时验证一些属性值。代码清单 3.7 所示的测试代码表明了如何手动检查一个 Item 实例的完整性。

代码清单3.7 测试一个Item实例是否存在约束冲突

路径: /examples/src/test/java/org/jpwh/test/simple/ModelOperations.java

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```



```
Item item = new Item();
item.setName("Some Item");
item.setAuctionEnd(new Date());
```

一条验证错误消息: 拍卖结束
日期不是未来的时间点!

```
Set<ConstraintViolation<Item>> violations = validator.validate(item);
assertEquals(1, violations.size());

ConstraintViolation<Item> violation = violations.iterator().next();
String failedPropertyName =
    violation.getPropertyPath().iterator().next().getName();

assertEquals(failedPropertyName, "auctionEnd");

if (Locale.getDefault().getLanguage().equals("en"))
    assertEquals(violation.getMessage(), "must be in the future");
```

我们打算详细解释这段代码, 而是留待你自己探究。你很少会编写此类验证代码; 大多数时候, 这部分内容都是由你的用户界面和持久化框架来自动处理的。因此, 在选择一种 UI 框架时探究 Bean 验证集成是很重要的, 比如, JSF 第 2 版以及较新版本会与 Bean 验证自动集成。

Hibernate 是所有 JPA 提供程序都需要的对象, 如果类路径上提供了库并提供了以下特性的话, 它也会自动集成 Hibernate 验证器:

- 在将实例传递给 Hibernate 用于存储之前, 你不必手动验证实例。
- Hibernate 会识别持久化域模型类上的约束, 并在数据库插入或更新操作之前触发验证。当验证失败时, Hibernate 会抛出一个 `ConstraintViolationException`, 其中包含失败的详情, 并将该异常提供给调用持久化管理操作的代码。
- 用于自动 SQL 模式生成的 Hibernate 工具集可以理解许多约束, 并且自动生成相当于 SQL DDL 的约束。例如, `@NotNull` 注解会转译成一个 SQL NOT NULL 约束, 而 `@Size(n)` 规则会定义 `VARCHAR(n)` 类型列中字符的数量。

可在 `persistence.xml` 配置文件中 使用 `<validation-mode>` 元素控制 Hibernate 的这一行为。默认模式是 `AUTO`, 因此 Hibernate 将仅在找到运行中应用程序类路径上的 Bean 验证提供程序(比如 Hibernate 验证器)时才会进行验证。而使用 `CALLBACK` 模式, 验证将总是发生, 并且如果忘记捆绑一个 Bean 验证提供程序, 则会得到一个部署错误。 `NONE` 模式会禁用 JPA 提供程序的自动验证。

在本书的后续内容中, 你将再次看到 Bean 验证注解; 你还会在示例代码包中找到它们。目前我们可以编写许多与 Hibernate 验证器有关的代码, 但这样做只不过是重复该项目优秀参考指南中的已有代码。浏览一下, 找出更多与像验证组和元数据 API 这样的功能有关的内容, 以便进一步了解约束。

Java 持久化和 Bean 验证标准积极引入了注解。其专家组已经意识到 XML 部署描述符在特定情形下的优势, 尤其适用于会随着每次部署而变更的配置元数据。

3.3.3 使用 XML 文件外部化元数据

可以使用 XML 描述符元素替换或重写 JPA 中的每一个注解。换句话说，如果不愿意，或者无论出于什么原因，将映射元数据从源代码中分离出来对你的系统设计有好处的话，可以不必使用注解。

1. 使用 JPA 的 XML 元数据

代码清单 3.8 显示了一个用于特定持久化单元的 JPA XML 描述符。

代码清单3.8 包含一个持久化单元的映射元数据的JPA XML描述符

首先是全局元数据

```
<entity-mappings
version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">
  <persistence-unit-metadata>
    <xml-mapping-metadata-complete/>
    <persistence-unit-defaults>
      <delimited-identifiers/>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
  <entity class="org.jpwh.model.simple.Item" access="FIELD">
    <attributes>
      <id name="id">
        <generated-value strategy="AUTO"/>
      </id>
      <basic name="name"/>
      <basic name="auctionEnd">
        <temporal>TIMESTAMP</temporal>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

忽略 XML 文件中所有的注解以及所有映射元数据

一些默认设置

避开所有 SQL 列、表和其他名称：例如，如果 SQL 名称是关键字的话(比如 USER)

如果将此描述符放置在该持久化单元类路径上的 META-INF/orm.xml 文件中的话，则 JPA 提供程序会自动选取此描述符。如果选择使用另一个名称或者多个文件的话，则必须 在你的 META-INF/persistence.xml 文件中修改该持久化单元的配置：

路径：/model/src/main/resources/META-INF/persistence.xml

```
<persistence-unit name="SimpleXMLCompletePU">
...
</persistence-unit>
```

```

    <mapping-file>simple/Mappings.xml</mapping-file>
    <mapping-file>simple/Queries.xml</mapping-file>
    ...
</persistence-unit>

```

如果包含<xml-mapping-metadata-complete>元素，则 JPA 提供程序会在这个持久化单元中忽略域模型类上的所有注解，并且仅依赖在 XML 描述符中定义的映射。可以(在个例子中不必这样做，会显得多余)使用<metadata-complete="true"/>在实体级别启用这一依赖。如果启用依赖，则 JPA 提供程序会假定你在 XML 中映射了该实体的所有属性，并且它应该为这个特定实体忽略所有注解。

相反，如果不希望忽略，而是重写注解元数据，则不要将 XML 描述符标记为“完成”，并对类和属性命名以便重写：

```

<entity class="org.jpwh.model.simple.Item">
  <attributes>
    <basic name="name">
      <column name="ITEM_NAME"/>
    </basic>
  </attributes>
</entity>

```

← 重写 SQL 列名称

此处你将 name 属性映射到 ITEM_NAME 列；默认情况下，该属性会映射到 NAME 列。Hibernate 现在会忽略来自 Item 类的 name 属性上 javax.persistence.annotation 和 org.hibernate.annotations 包的任何已有注解。但 Hibernate 不会忽略 Bean 验证注解，并且仍旧会将它们应用于自动验证和架构生成！Item 类上的所有其他注解也会识别。注意没有在这个映射中指定访问策略，所以会使用字段访问或访问器方法，这取决于@Id 注解在 Item 中的位置(我们将在第 4 章中再次详细地探讨这个问题)。

本书不会过多介绍与 JPA XML 描述符有关的内容。这些文件的语法与 JPA 注解语法完全相同，所以你编写它们应该不会有任何问题。我们将专注于重要的部分：映射策略。用来描述元数据的语法是次要的。

遗憾的是，与 Java EE 世界中的许多其他架构一样，JPA orm_2_0.xsd 不允许供应商扩展。你不能在 JPA XML 映射文件中使用另一个命名空间的元素和属性。因而，使用供应商扩展和 Hibernate 原生功能需要借助另一种 XML 语法。

2. Hibernate XML 映射文件

在 JDK 5 引入注解之前，原生 Hibernate XML 映射文件格式是原始的元数据选项。按照惯例，你要使用.hbm.xml作为后缀命名这些文件。代码清单 3.9 显示了一个基础 Hibernate XML 映射文件。

代码清单3.9 以Hibernate原生的XML语法编写的元数据文件
 路径：/model/src/main/resources/simple/Native.hbm.xml

```

<?xml version="1.0"?>
<hibernate-mapping

```




① 元数据是在<hibernate-mapping>根元素上声明的。像包名称这样的属性和默认访问会在此文件中应用到所有映射。可以随意包括许多实体类映射。

注意，此 XML 文件为所有元素声明了一个默认 XML 命名空间；这是 Hibernate 5 中的一个新选项。如果有基于 Hibernate 4 的已有映射文件或者以 XML 文件类型声明的较早文件，可以继续使用它们。

尽管在一个映射文件中使用多个<class>元素声明用于多个类的映射是可行的，但许多较早的 Hibernate 项目都是通过为每一个持久化类使用一个映射文件来组织的。其便利性在于为该文件赋予与映射类相同的名称和包：例如，将 my/model/Item.hbm.xml 用于 my.model.Item 类。

Hibernate XML 文件中的类映射是一个“完全”映射；也就是说，该类的任何其他映射元数据，不管是在注解中还是在 JPA XML 文件中，都将在启动时触发一个“重复映射”的错误。如果在 Hibernate XML 文件中映射一个类，则此声明必须包括所有映射详情。你不能重写单独的属性或者扩展一个已有映射。此外，还必须在一个 Hibernate XML 文件中列出并映射一个实体类的所有持久化属性。如果不映射属性，则 Hibernate 会认为它处于临时状态。将其与 JPA 映射做一下比较，JPA 映射中的一个@Entity 注解就会让类的所有属性持久化。

Hibernate 原生 XML 文件不再是声明项目大部分 ORM 元数据的主要选择。大多数工程师现在都选用注解。原生 XML 元数据文件主要用于获得对特殊 Hibernate 功能的访问，这些功能在注解中不可用或者更易于在 XML 文件中维护(例如，出于是依赖于部署的配置元数据的原因)。你不需要在 Hibernate XML 映射文件中使用任何<class>元素。因此，这些文件中所有的元数据对于持久化单元都可以是全局的，比如外部化(甚至原生 SQL)查询字符串、自定义的类型定义、用于特定 DBMS 产品的辅助 SQL DDL、动态持久化上下文过滤器等。

在随后探讨这些高级和原生的 Hibernate 特性时，我们将介绍如何在 Hibernate XML 文件中声明它们。如前所述，你的精力应该专注于理解映射策略的本质，而我们大多数示例都会使用 JPA 和 Hibernate 注解来表述这些策略。

目前为止我们所描述的方法会假定所有 ORM 元数据在开发(或部署)时都是已知的。假定在应用程序启动之前不知道一些信息。你可以在运行时编程操作该映射元数据吗？我们还提到过用于访问持久化单元详情的 JPA 元数据 API。它如何发挥作用，并且何时生效呢？

3.3.4 在运行时访问元数据

JPA 规范提供了用于访问持久化类元模型的编程接口，该 API 有两种类型。第一个选项在本质上更为动态，并且类似于基础 Java 反射。第二个选项是一个静态元模型，通常是由 Java 6 注解处理器所生成的。对于这两个选项来说，访问都是只读的；你不能在运行时修改元数据。

Hibernate 还提供了一个原生元模型 API，它支持读取和写入访问以及与 ORM 有关的更多详情。本书不会介绍这一原生 API(可以在 `org.hibernate.cfg.Configuration` 中找到它)，因为它已经过时了，而编写本书时还没有出现替代的 API。请参考 Hibernate 文档以获得此特性的最新更新。

1. Java 持久化中的动态元模型 API

有时候——例如，当希望编写一些自定义验证或者通用 UI 代码时——你会想要获得对一个实体的持久化属性的编程访问。你会想要知道域模型动态持有哪些持久化类和属性。代码清单 3.10 中的代码显示了如何使用 Java 持久化接口读取元数据。

代码清单3.10 使用元模型API获得实体类型信息

路径: `/examples/src/test/java/org/jpwh/test/simple/AccessJPAMetamodel.java`

```
Metamodel mm = entityManagerFactory.getMetamodel();

Set<ManagedType<?>>managedTypes = mm.getManagedTypes();
assertEquals(managedTypes.size(), 1);

ManagedType itemType = managedTypes.iterator().next();
assertEquals(
    itemType.getPersistenceType(),
    Type.PersistenceType.ENTITY
);
```

可以从通常每个数据源的应用程序中只有一个实例的 `EntityManagerFactory` 获得元模型，或者更方便的方法是，通过调用 `EntityManager#getMetamodel()` 来获得元模型。该托管类型集包含与所有持久化实体和内置类有关的信息(我们将在下一章进行介绍)。在这个示例中，只有一个对象：`Item` 实体。这就是你如何更深入探究并找出更多与每个属性有关的内容的方式。请参见代码清单 3.11。

代码清单3.11 使用元模型API获得实体属性信息

路径: /examples/src/test/java/org/jpwh/test/simple/AccessJPAMetamodel.java

```
SingularAttribute nameAttribute =
    itemType.getSingularAttribute("name"); ← ① 实体属性
assertEquals(
    nameAttribute.getJavaType(),
    String.class
);
assertEquals(
    nameAttribute.getPersistentAttributeType(),
    Attribute.PersistentAttributeType.BASIC
);
assertFalse(
    nameAttribute.isOptional() ← 非 NULL
);
SingularAttribute auctionEndAttribute =
    itemType.getSingularAttribute("auctionEnd"); ← ② 实体属性
assertEquals(
    auctionEndAttribute.getJavaType(),
    Date.class
);
assertFalse(
    auctionEndAttribute.isCollection()
);
assertFalse(
    auctionEndAttribute.isAssociation()
);
```

该实体的属性是使用字符串来访问的: **name①**和 **auctionEnd②**。这显然不是类型安全的, 并且如果修改属性的名称, 则此代码会被破坏并废弃。该字符串不会自动包含在 IDE 的重构操作中。

JPA 还提供一种静态类型安全的元模型。

2. 使用一个静态元模型

Java(至少到版本 8 之前)没有对属性的最优支持。你无法以类型安全的方式访问一个 bean 的字段或访问器方法——只能通过其名称, 使用字符串进行访问。这对于使用 JPA 标准查询尤为不便, 该查询对于基于字符串的查询语言是一种类型安全的可替代方式。下面列举一个例子:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Item> query =
    cb.createQuery(Item.class);
Root<Item> fromItem = query.from(Item.class);
query.select(fromItem);
List<Item> items =
    entityManager.createQuery(query)
```

此查询等同于“select i from Item i.”


```

        .getResultList();

assertEquals(items.size(), 2);

```

这个查询会返回数据库中所有的商品项；这里有两件商品。如果现在希望限定这个结果，并且仅返回具有特定名称的商品，则必须使用 `like` 表达式，以使用在参数中设置的模式来比较每件商品的 `name` 属性：

```

Path<String> namePath = fromItem.get("name");
query.where(
    cb.like(
        namePath,
        cb.parameter(String.class, "pattern")
    )
);
items =
    entityManager.createQuery(query)
        .setParameter("pattern", "%some item%")
        .getResultList();
assertEquals(items.size(), 1);
assertEquals(items.iterator().next().getName(), "This is some item");

```

“模式 where i.name like”

必须是用于 `like()` 运算符的 `Path<String>!`

通配符!

注意 `namePath` 查找是如何需要 `name` 字符串的。这就是标准查询的类型安全失败的位置。可以用你的 IDE 重构工具重命名 `Item` 实体类，且该查询仍然有效。但只要你触及 `Item#name` 属性，则手动调整就是必要的行为。幸运的是，可以在测试失败时捕获到此错误。

一种更好的方式是类型安全的静态元模型，该方式对于在编译时而非运行时进行重构和检测不匹配很安全：

```

query.where(
    cb.like(
        fromItem.get(Item_.name),
        cb.parameter(String.class, "pattern")
    )
);

```

静态 `Item_` 元模型!

此处特殊的类是 `Item_`；注意其中的下划线。这个类是一个元数据类，并且，此处列出了该 `Item` 实体类的所有属性：

```

@javax.persistence.metamodel.StaticMetamodel(Item.class)
public abstract class Item_ {

    public static volatile SingularAttribute<Item, Long> id;
    public static volatile SingularAttribute<Item, String> name;
    public static volatile SingularAttribute<Item, Date> auctionEnd;
}

```

可以手动编写这个类，或者按照这个 API 设计者的意图，让其通过 Java 编译器的注解处理工具(annotation processing tool, apt)来自动生成。Hibernate JPA2 元模型生成器(Hibernate 套件的一个不同子项目)使用了这一扩展点。其唯一的目标是从你的托管持久化

类中生成静态元模型类。可下载其 JAR 文件并将该文件集成到你的 IDE 中(或者你的 Maven 构造中,就像本书的示例代码中所做的那样)。无论何时编译(或者修改,取决于 IDE)该 Item 实体类并生成合适的 Item_元数据类,它都会自动运行。

什么是注解处理工具(apt)?

Java 包含命令行工具 apt, 或者称为注解处理工具, 会根据源代码中的注解找到并执行注解处理器。注解处理器会使用反射 API 来处理程序注解(JSR 175)。apt API 提供了一个编译时的源文件和程序的只读视图来对 Java 类型系统建模。注解处理器可能首先会生成新的源代码和文件, 之后 apt 可以将其与最初的源一起编译。

尽管在前面几节内容中你已经看到了一些映射结构, 但到目前为止我们还没有介绍更为复杂的类和属性映射。你现在应该决定在项目中希望使用哪种映射元数据的策略——我们推荐使用注解, 并且仅在必要时使用 XML——之后可以在第 4 章中了解到更多与类和属性映射有关的内容。

3.4 本章小结

- 你已经实现了免受像日志记录、授权和事务分界这样的任何横切关注点所影响的持久化类; 你的持久化类仅在编译时依赖 JPA。即便是与持久化相关的关注点都不应该渗漏到域模型实现中。
- 如果希望独立且很容易地执行和测试你的业务对象, 则透明持久化就很重要。
- 你已经学习可用于 POJO 和 JPA 实体编程模型的最佳实践和需求, 以及它们与传统 JavaBean 规范有哪些共同的概念。
- 你已经做好编写更复杂映射的准备, 可能会使用 JDK 注解或 JPA/Hibernate XML 映射文件的组合。

映射策略

本部分完全与实际的 ORM 有关，涉及从类和属性到表和列。第 4 章从常规类和属性映射开始介绍，以及阐释如何映射细粒度 Java 域模型。接着在第 5 章中，你将看到如何映射基础属性和可内嵌的组件，以及如何控制 Java 和 SQL 类型之间的映射。在第 6 章中，你要使用四个基础继承映射策略来将实体的继承性层次结构映射到数据库；还要映射多态关联。第 7 章完全与映射集合和实体关联有关：你要映射持久化集合、基础和可内嵌类型的集合、以及简单多对一和一对多的实体关联。第 8 章会更深入探讨高级实体关联映射，比如映射一对一实体关联、一对多的映射选项以及多对多和三元实体关系。最后，如果需要在现有应用程序中引入 Hibernate，或者必须使用遗留数据库架构和手动编写的 SQL，则会发现第 9 章有最密切相关的内容。我们还将在这一章中讨论与用于架构生成的自定义 SQL DDL 有关的内容。

在阅读完本书的这一部分之后，你将做好甚至能快速创建最复杂映射和使用正确策略的准备。你将理解如何解决继承映射的问题以及如何映射集合和关联。你还将能够为集成任何已有数据库架构或应用程序而调整和自定义 Hibernate。

4.1 理解域模型和类类型

当尝试建模时，你将会注意到模型的一个区别：有些类型看起来更为重要，它们表示了最重要的业务对象（重要的对象，更接近其本身的原型）。例子有 Item、Category 和 User 类（这些是你希望存储在数据库中的实体对象如图 3.3 中所示的域模型的视图）。域模型中呈现的其他类型，比如 Address、STRING 和 INTEGER，看起来不那么重要。在这一节中，我们将介绍使用细粒度域模型并在实体和值类型之间进行区分意味着什么。

4.1.1 细粒度域模型

Hibernate 的一个主要目的就是支持细粒度的域模型。这是使用 POJO 的一个原因。简单地说，细粒度意味着类比表要多。

例如，域模型中的一个用户可能有一个家庭住址。在数据库中，可以使用具有 HOME_STREET、HOME_CITY 和 HOME_ZIPCODE 列的单个 USERS 表（还记得我们在 1.2.1 节中讨论过的 SQL 类型问题吗？）。

本章内容简介：

- 理解实体和值类型的概念
- 用标识映射实体类
- 控制实体级的映射选项

这一章会介绍一些基础映射选项并说明如何将实体类映射到 SQL 表。我们将展示并探讨如何能够处理数据库标识和主键，以及如何使用各种其他元数据设置来自定义 Hibernate 加载和存储域模型类实例的方式。所有的映射示例都使用 JPA 注解。不过，首先我们要定义实体和值类型之间的本质区别，并阐释你应该如何着手处理域模型的对象/关系映射。

JPA 2 中主要的新特性

可以使用 `persistence.xml` 配置文件中的 `<delimited-identifiers>` 元素在全局范围对所生成 SQL 语句中的所有名称进行转义。

4.1 理解实体和值类型

当查看域模型时，你将会注意到类之间的一个区别：有些类型看起来更为重要，它们表示了最重要的业务对象(这里的对象一词就是其本身的意思)。例子有 `Item`、`Category` 和 `User` 类：这些都是你要尝试表示的真实环境中的实体(回顾图 3-3 中该示例域模型的视图)。域模型中呈现的其他类型，比如 `Address`、`STRING` 和 `INTEGER`，看起来不那么重要。在这一节中，我们将介绍使用细粒度域模型并在实体和值类型之间进行区分意味着什么。

4.1.1 细粒度域模型

Hibernate 的一个主要目标是支持细粒度的富域模型，这是我们使用 POJO 的一个原因。简单来说，细粒度意味着类比表要多。

例如，域模型中的一个用户可以有一个家庭地址。在数据库中，可以使用具有 `HOME_STREET`、`HOME_CITY` 和 `HOME_ZIPCODE` 列的单个 `USERS` 表(还记得我们在 1.2.1 节中探讨过的 SQL 类型问题吗？)。

在该模型中，可以使用相同的方法，以便将地址表示为 `User` 类的三个字符串值属性。但更好的做法是使用一个 `Address` 类对其建模，其中 `User` 具有一个 `homeAddress` 属性。这个域模型实现了内聚性的提升以及更好的代码重用，并且它比具有不灵活类型系统的 SQL 更易于理解。

JPA 强调的是用于实现类型安全和行为的细粒度类的有效性。例如，许多人将电子邮件地址建模为 `User` 的字符串值属性。一种更为先进的方式是定义一个 `EmailAddress` 类，它会添加较高级别的语义和行为——它可以提供一个 `prepareMail()` 方法(不应该有 `sendMail()` 方法，因为你不希望自己的域模型类依赖邮件子系统)。

这种粒度问题会让我们思考 ORM 中具有核心重要性的一个区别。在 Java 中，所有的类都是平等的——所有实例都有其自己的标识和生命周期。当引入持久化时，一些实例可能就没有其自己的标识和生命周期了，而是依赖其他的标识和生命周期。我们来看一个示例。

4.1.2 定义应用程序概念

有两个人生活在同一幢住所内，并且他们都在 `CaveatEmptor` 上注册了用户账户。不妨称呼他们为 John 和 Jane。

`User` 的一个实例代表一个账户。由于你希望单独加载、保存和删除这些 `User` 实例，因此 `User` 是一个实体类而非值类型。找出实体类很简单。

`User` 类具有一个 `homeAddress` 属性；它是与 `Address` 类的一种关联关系。这两个 `User` 实例都有对相同 `Address` 实例的运行时引用吗？或者每个 `User` 实例有其自己的 `Address` 引用吗？John 和 Jane 生活在同一幢住所内有关系吗？

在图 4-1 中，可以看到两个 `User` 实例如何共享单个 `Address` 实例(这是一个 UML 对象图而非类图)。如果 `Address` 应当支持共享运行时引用的话，则它就是一个实体类型。`Address` 实例有其自己的生命周期，你不能在 John 移除其 `User` 账户时将其删除——Jane 可能仍然具有对 `Address` 的引用。

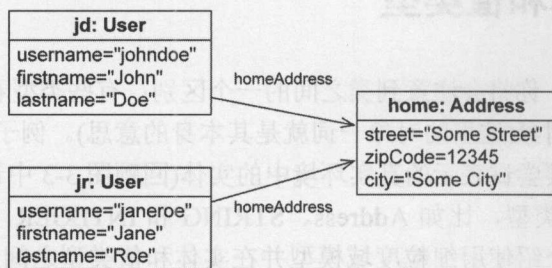


图 4-1 两个 `User` 实例具有对单个 `Address` 的引用

现在来看看备选模型，其中每个 `User` 都有其自己的 `homeAddress` 实例的引用，如图 4-2 所示。在这种情况下，可以让 `Address` 的实例依赖 `User` 的实例：将它作为值类型。当 John 移除其 `User` 账户时，可以安全地删除他的 `Address` 实例。没有其他人保留引用了。

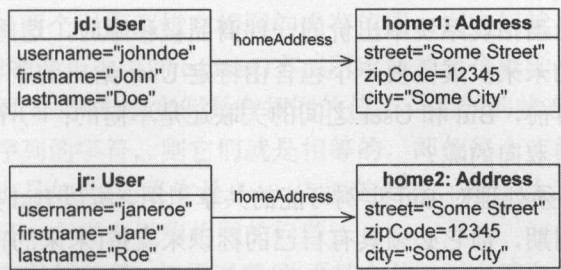


图 4-2 两个 User 实例都有其自己独立的 Address

因此，我们得出了以下本质区别：

- 可以使用实体类型实例的持久化标识检索它：例如，一个 User、Item 或 Category 实例。实体实例的引用(JVM 中的一个指针)会持久化为数据库中的一个引用(一个受外键约束的值)。实体实例具有其自己的生命周期；它可以独立于所有其他实体存在。你要将所选择的域模型类映射为实体类型。
- 值类型的一个实例不具有持久化标识符属性；它属于一个实体实例。其生命周期绑定到持有它的实体实例。值类型实例不支持共享引用。最明显的值类型就是所有 JDK 定义的类，比如 String、Integer 甚至基元。你还可以将自己的域模型类映射为值类型：例如，Address 和 MonetaryAmount。

如果阅读 JPA 规范，就会发现相同的概念。但 JPA 中的值类型称为基础属性类型或者可内嵌的类。我们将在第 5 章回过头来探讨这部分内容；我们首要的关注点是实体。

识别域模型中的实体和值类型并非一项特别的任务，而是遵循一个特定过程。

4.1.3 区分实体和值类型

你可能发现，将原型(一种 UML 扩展性机制)信息添加到 UML 类图很有帮助，这样就可以立即识别出实体和值类型。这一做法还会强制你思考所有类的这种区别，这是得到最优映射和可良好执行持久化层的第一步。图 4-3 显示了一个示例。

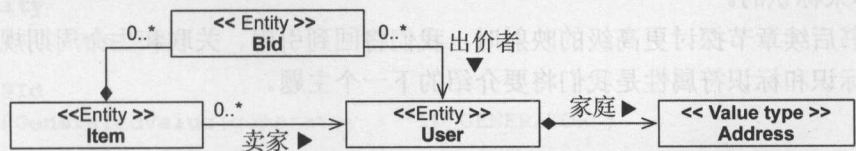


图 4-3 用于实体和值类型的原型图解

Item 和 User 类显然是实体。它们有其自己的标识，其实例具有来自许多其他实例的引用(共享引用)，并且有独立的生命周期。

将 Address 标记为一个值类型也很容易：单个 User 实例引用一个特定 Address 实例。你清楚这一点，因为该关联已经创建为一个组合，其中已经让 User 实例完全负责所引用的 Address 实例的生命周期。因此，Address 实例不能被其他实例所引用，并且不需要自己的标识。

Bid 类可能会是一个问题。在面向对象的建模中，这标记为一个组合(Item 和 Bid 之间具有菱形关系的关联)。因此，Item 是其 Bid 实例的持有者，并且持有一组引用。最初，这

看起来是合理的，因为当拍卖系统中出价的一件商品被移除时，这些出价就没有意义了。

但如果该域模型的将来扩展需要一个包含由特定 User 给出的所有出价的 User#bids 集合，又该怎么办呢？目前，Bid 和 User 之间的关联还是单向的；一个 Bid 具有一个出价者引用。但如果该关联是双向的呢？

在该情况下，你必须处理对 Bid 实例可能的共享引用，所以 Bid 类需要成为一个实体。它具有依赖性的生命周期，但它必须具有自己的标识来支持(未来的)共享引用。

你会经常看到这类混合行为；但你的第一反应应该是让一切都成为值类型的类，并且仅在绝对必要的时候才提升该类为一个实体。试着简化你的关联：例如，持久化集合常常会添加复杂性，而不提供任何好处。相较于映射 Item#bids 和 User#bids 集合，可以编写查询来获得 Item 的所有出价以及某个特定 User 的出价。UML 图中的关联从 Bid 指向 Item 和 User，它是单向的，而不是反向的。这样 Bid 类上的原型就会是<<Value type>>。我们将在第 7 章再次回到这一点上进行介绍。

接下来，使用你的域模型图，并且为所有的实体和值类型实现 POJO。你必须注意三件事情：

- 共享引用——当编写 POJO 类时，要避免对值类型实例的共享引用。例如，确保只有一个 User 可以引用一个 Address。可以使用非公共的 setUser() 方法让 Address 不可变，并且使用一个公共的具有一个 User 参数的构造函数强制实施该关系。当然，你仍然需要一个不含参数、可能还受保护的构造函数，就像我们在第 3 章中所探讨过的，这样 Hibernate 就能创建一个实例。
- 生命周期依赖——如果一个 User 被删除，则其 Address 依赖也必须删除。持久化元数据将包含用于所有这些依赖的级联规则，这样 Hibernate(或数据库)就能负责移除已废弃的 Address。你必须设计应用程序过程和用户界面来遵循及预期这样的依赖——相应地编写域模型 POJO。
- 标识——几乎在所有情况下，实体类都需要一个标识符属性。值类型类(当然还有像 String 和 Integer 这样的 JDK 类)不具有标识符属性，因为实例是通过持有它的实体来标识的。

在本书后续章节探讨更高级的映射时，我们将回到引用、关联和生命周期规则的主题上。对象标识和标识符属性是我们将要介绍的下一个主题。

4.2 映射具有标识的实体

在我们介绍一个实体类示例及其映射之前，映射具有标识的实体需要你理解 Java 标识和相等性。在此之后，我们就能够进行更深入的探究以及选择一个主键，配置键生成器，并最终探讨标识符生成器策略。首先，在我们探讨像数据库标识这样的术语以及 JPA 管理标识的方式之前，理解 Java 对象标识和对象相等性之间的区别至关重要。

4.2.1 理解 Java 标识和相等性

Java 开发人员理解 Java 对象标识和相等性之间的区别。对象标识(==)是由 Java 虚拟机

定义的一个概念。如果两个引用都指向相同的内存位置，则它们就是完全相同的。

另一方面，对象相等性是由类的 `equals()` 方法所定义的一个概念，有时候也称为等效性。等效性意味着两个不同(非等同)的实例具有相同的值——相同的状态。如果 `String` 的两个不同实例表示了相同序列的字符，则它们就是相等的，即使每个实例都在虚拟机的内存空间中具有自己的位置也是如此(如果你是 Java 方面的专家，则会认为 `String` 是一个特例。那么，假设我们使用另一个类来说明相同的问题)。

持久化让此情况变得复杂了。使用对象/关系持久化，一个持久化实例就是数据库某个表(或多个表)中的一个特定行(或多个行)的内存表示。与 Java 标识和相等性一起，我们要定义数据库标识。你现在有三种区分引用的方法：

- 如果对象在 JVM 中占据了相同的内存位置，则它们就是完全相同的。这可以使用 `a == b` 运算符来验证。这一概念称为对象标识。
- 如果对象具有相同的状态，比如通过 `a.equals(Object b)` 方法进行定义，则它们就是相等的。未显式重写这个方法的类会继承由 `java.lang.Object` 定义的实现，它会使用 `==` 来比较对象标识。这一概念称为对象相等性。
- 如果存储在一个关系型数据库中的对象共享相同的表和主键值，则它们就是完全相同的。这一映射到 Java 领域的概念称为数据库标识。

现在我们需要查看数据库标识如何与对象标识相关以及如何在映射元数据中表述数据库标识。作为一个示例，你要映射一个域模型的实体。

4.2.2 第一个实体类和映射

在第3章中，我们并非完全忠于事实：`@Entity` 注解不足以映射一个持久化类。你还需要一个 `@Id` 注解，如代码清单 4.1 所示。

代码清单4.1 具有一个标识符属性的已映射Item实体类

路径：/model/src/main/java/org/jpwh/model/simple/Item.java

```
@Entity
public class Item {

    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    protected Long id;

    public Long getId() {          ← 可选但有用
        return id;
    }
}
```

这是最基础的实体类，使用 `@Entity` 注解标记为“可持久化”，并且带有一个用于数据库标识符属性的 `@Id` 映射。这个类默认会映射到数据库架构中一个名为 `ITEM` 的表。

每个实体类都必须具有一个 `@Id` 属性；它是 JPA 将数据库标识公开给应用程序的方式。我们不在图表中显示该标识符属性；假定每个实体类都有一个该属性。在我们的示例中，总是将该标识符属性命名为 `id`。这对于你自己的项目来说也是很好的做法；为所有的域模

型实体类使用相同的标识符属性名称。如果不指定其他内容，那么这个属性就会映射到数据库架构的 ITEM 表中名为 ID 的主键列。

在加载和存储条目时，Hibernate 会使用该字段而非获取方法或设置方法来访问标识符属性值。由于 @Id 在一个字段上，因此现在 Hibernate 默认会将类的每个字段用作持久化属性。JPA 中的规则是：如果 @Id 在一个字段上，则 JPA 提供程序将直接访问这个类的字段，并且默认会将所有字段看成持久化状态的一部分。在本章稍后的内容中，你将看到如何重写它——以我们的经验来看，字段访问通常是最佳选择，因为它允许更自由地进行访问器方法设计。

你应该将一个(公共)获取方法用于标识符属性吗？应用程序通常使用数据库标识符作为特定实例的一个便利手柄，即便在持久化层之外也是这样做的。例如，Web 应用程序将搜索结果的界面作为一系列汇总显示给用户是很常见的情况。当用户选择某特定元素时，应用程序可能需要检索选中的项，并且通常会为此目的使用按标识符的查找——即使是在依赖 JDBC 的应用程序中，你大概也已经像这样使用标识符。

你应该使用设置方法吗？主键值永远不会发生变化，因此你不应该允许对标识符属性值进行修改。Hibernate 不会更新主键列，并且你不应该公开一个实体上的公共标识符设置方法。

标识符属性的 Java 类型，即前一个示例中的 `java.lang.Long`，取决于 ITEM 表的主键列类型以及键值是如何生成的。到此，我们就该从总体上谈一下 @GeneratedValue 注解和主键了。

4.2.3 选择一个主键

实体的数据库标识符会映射到一些表主键，因此我们首先介绍关于主键的一些背景知识，而无须担心映射的问题。回顾一下你是如何标识实体的。

候选键是可以用来标识一个表中特定行的一列或几列。要成为主键，候选键必须满足以下要求：

- 所有候选键的值都绝不能为空。你不能用未知的数据来标识列，并且关系模型中不存在空值。一些 SQL 产品允许使用可以为空的列来定义(构成)主键，所以你必须小心谨慎。
- 候选键列的值是用于所有行的唯一值。
- 候选键列的值绝不能变更；它是不可变的。

主键必须是不可变的吗？

关系模型定义了一个候选键必须是唯一且不可再分的(该键属性的子集具有唯一属性)。除此之外，选择一个候选键作为主键是偏好问题。但 Hibernate 预期候选键在用作主键时是不可变的。Hibernate 不支持使用 API 更新主键值；如果试图绕过这一要求，则会在使用 Hibernate 的缓存和脏检查引擎时遇到问题。如果数据库模式依赖可更新的主键(以及可能使用了 ON UPDATE CASCADE 外键约束)，那么必须在将其与 Hibernate 一起使用之前修改该模式。

如果一个表只有一个标识属性，那么根据定义，它将变成主键。但几列或者列的几种组合可以满足用于某个特定表的这些属性；可以在候选键之间进行选择以便决定用于这个表的最佳主键。如果候选键的值确实是唯一的(但可能不是不可变的)，那么你应该声明这些候选键不作为主键选择并成为数据库中的唯一键。

许多遗留的 SQL 数据模型都使用自然主键。自然键是具有业务含义的键：由于其业务语义，其值是唯一的一个或一组属性。自然键的例子包括美国社会安全号码以及澳大利亚税务档案号码。区别自然键很简单：如果一个候选键属性具有数据库上下文之外的含义，则它就是一个自然键，无论其是否是自动生成的。考虑一下应用程序的用户：如果他们在谈论和使用应用程序时涉及一个键属性，那么这个键就是一个自然键：“你能将条目 #123-abc 的图片发给我吗？”

经验表明，自然主键最后通常会造成问题。一个好的主键必须是唯一的、不可变的，并且绝不能为空。几乎没有实体属性满足这些要求，并且有些还不能被 SQL 数据库有效索引(尽管这是一个实现细节，并且不应该是选择一个特定键的决定因素)。此外，你应该确定，一个候选键定义绝不能在数据库的生命周期中发生变化。变更一个主键的值(甚或定义)以及引用它的所有外键会是一项让人绝望的任务。应该期望你的数据库架构能够有效运行数十年，即使你的应用程序不会存在这么久。

再者，你通常只能通过将几列合并成一个组合自然键来得到自然候选键。尽管这些组合键绝对适合某些架构部件(比如多对多关系中的链接表)，但这些组合键也可能让维护、特定查询以及架构演化更为复杂。我们将在 9.2.1 节中探讨组合键。

基于这些原因，我们强烈建议你添加合成标识符，也称为代理键。代理键不具有业务含义——它们具有数据库或应用程序生成的唯一值。理想情况下，应用程序用户不会看见或涉及这些键值；它们是系统内部的一部分。在没有候选键这样的常见情形下，引入一个代理键也是合适的。换句话说，就算仅仅出于此目的考虑，模式中(几乎)每一个表都应该有一个专用的代理主键。

有许多众所周知的方法可以生成代理键值。前面提到的 `@GeneratedValue` 注解就是配置生成代理键值的一种方法。

4.2.4 配置键生成器

需要 `@Id` 注解来标记一个实体类的标识符属性。不在该属性的旁边使用 `@GeneratedValue` 的话，JPA 提供程序会假定在保存实例之前，你将负责创建和指定标识符值。我们将其称为应用程序指定的标识符。在处理一个遗留数据库和/或自然主键时，就必须手动指定实体标识符。我们将在 9.2.1 节中专门介绍更多与此类映射有关的内容。

当保存实体实例时，通常你会希望系统生成一个主键值，因此你会在 `@Id` 旁边编写 `@GeneratedValue` 注解。JPA 使用 `javax.persistence.GenerationType` 枚举标准化了几种值生成策略，可以使用 `@GeneratedValue(strategy = ...)` 进行选择：

- `GenerationType.AUTO`——Hibernate 会选择一种合适的策略，询问你已配置数据库的哪种 SQL 方言是最佳的。这等同于使用不进行任何设置的 `@GeneratedValue()`。

- `GenerationType.SEQUENCE`——Hibernate 预期(并且会创建, 如果使用工具的话)你的数据库中存在一个名为 `HIBERNATE_SEQUENCE` 的序列。该序列会在每个 `INSERT` 之前被单独调用, 以生成顺序数字值。
- `GenerationType.IDENTITY`——Hibernate 预期(并且会在表 DDL 中创建)一个特殊的自增长主键列, 该列会在数据库 `INSERT` 时自动生成一个数字值。
- `GenerationType.TABLE`——Hibernate 将在你的数据库架构中使用一个额外的表, 这个表会保存下一个数字主键值, 每行对应一个实体类。在 `INSERT` 之前, 该表将被读取并做相应更新。默认的表名称为 `HIBERNATE_SEQUENCES`, 它具有 `SEQUENCE_NAME` 和 `SEQUENCE_NEXT_HI_VALUE` 列(其内部实现使用了一种更为复杂但有效的高/低位生成算法; 稍后将对该算法进行更为详尽的介绍)。

尽管 `AUTO` 看起来很便利, 但你还需要更多的控制, 所以通常不应该依赖它, 而是明确配置一个主键生成策略。此外, 大多数应用程序都会使用数据库序列, 但你可能希望自定义名称及数据库序列的其他设置。因此, 相较于选取其中一种 JPA 策略, 我们推荐一种具有 `@GeneratedValue(generator = "ID_GENERATOR")` 的标识符映射, 如前一个示例所示。

这是一个指定的标识符生成器; 你现在可以独立地随意设置 `ID_GENERATOR` 配置, 无须担心你的实体类。

JPA 具有两个内置的注解, 可以用来配置指定生成器: `@javax.persistence.SequenceGenerator` 和 `@javax.persistence.TableGenerator`。有了这些注解, 就可以创建具有自己的序列和表名称的指定生成器。遗憾的是, 就像平常使用 JPA 注解一样, 你仅可以在一个(或许可能为空的)类的顶部使用它们, 而不是在 `package-info.java` 文件中使用。

Hibernate 特性

基于此原因, 并且由于 JPA 注解不能提供对完整 Hibernate 特性集的访问, 我们可以选用一种替代方式: 原生的 `@org.hibernate.annotations.GenericGenerator` 注解。它支持所有的 Hibernate 标识符生成器策略及其配置详情。尤其不同于受限的 JPA 注解, 可以在 `package-info.java` 文件中使用 Hibernate 注解, 通常是在与模型类相同的包中。代码清单 4.2 显示了一种推荐配置。

代码清单4.2 作为包级别元数据配置的Hibernate标识符生成器

路径: `/model/src/main/java/org/jpwh/model/package-info.java`

```
@org.hibernate.annotations.GenericGenerator(  
    name = "ID_GENERATOR",  
    strategy = "enhanced-sequence",           ← ①增强序列策略  
    parameters = {  
        @org.hibernate.annotations.Parameter(  
            name = "sequence_name",           ← ②序列名  
            value = "JPWH_SEQUENCE"  
        ),  
        @org.hibernate.annotations.Parameter(  
            name = "initial_value",           ← ③初始值  
            value = "1000"  
        )  
    }  
)
```

```
)  
})
```

这一特定于 Hibernate 生成器的配置具有以下优势：

- enhanced-sequence ①策略会生成序列化的数字值。如果 SQL 方言支持序列，那么 Hibernate 就会使用一个真实的数据库序列。如果 DBMS 不支持原生序列，那么 Hibernate 就会管理并使用一个额外的“序列表”，模拟序列的行为。这就赋予了你真正的可移植性：该生成器总是能在执行 SQL INSERT 之前被调用。例如，它不同于会在 INSERT 上生成一个随后必须返回给应用程序的值的自增长标识列。
- 可以配置 sequence_name②。Hibernate 要么会使用一个现有的序列，要么会在自动生成 SQL 架构时创建该序列。如果 DBMS 不支持序列，那么这将是该特殊“序列表”的名称。
- 可以从一个提供测试数据空间的 initial_value③开始着手处理。例如，当你的集成测试运行时，Hibernate 将来自测试代码的所有新数据插入值大于 1000 的标识符。在测试之前，你想要导入的所有测试数据都可以使用 1 到 999 的数值，并且可以在测试中引用固定标识符值：“加载 id 为 123 的条目并对它运行一些测试。”这是在 Hibernate 生成 SQL 架构和序列时应用的操作；它是一个 DDL 选项。

可以在所有的域模型类之间共享相同的数据库序列。在所有的实体类中指定 @GeneratedValue(generator = "ID_GENERATOR")没什么坏处。对于一个特定实体，主键值是否连续并不重要，只要它们在一个表中是唯一的就行。如果担心产生竞争，因为序列必须先于每一个 INSERT 被调用，那么请不要担心，我们将在第 20.1 节中探讨这一生成器配置的变体。

最后，你要使用 java.lang.Long 作为实体类中标识符属性的类型，它会完美映射到一个数值数据库序列生成器。你还可以使用 long 基元。其主要区别是在还未存储到数据库中的新条目上 someItem.getId()会返回什么：null 或 0。如果希望测试一个条目是否是新增的，那么 null 检验可能更易于其他阅读你代码的人理解。你不应该将另一种整数类型用于标识符，比如 int 或 short。尽管它们可以有效运行一段时间(甚至可能几年)，但随着你的数据库规模增大，你可能会受到其值域的限制。如果无间隔地每毫秒生成一个新的标识符，那么一个 Integer 可以有效运行几乎两个月，而一个 Long 则可以有效运行大约 3 亿年。

尽管推荐大多数应用程序使用，但代码清单 4.2 中所示的 enhanced-sequence 策略只是一个内置在 Hibernate 中的策略。

4.2.5 标识符生成器策略

以下是所有可用的 Hibernate 标识符生成器策略、其选项以及我们的使用推荐清单。如果现在不想阅读整个清单，则可以启用 GenerationType.AUTO 并检查 Hibernate 对你的数据库方言有哪些默认设置。这很可能是序列或标识——一个优秀但可能不是最有效或可移植的选项。如果需要一致的可移植行为，并且让标识符值可在 INSERT 之前使用，则可以使用 enhanced-sequence，如上一节中所述。这是一种可移植的、灵活且现代的策略，还为大型数据集提供了各种优化器。

在 INSERT 之前或之后生成标识符：区别是什么？

ORM 服务会试图优化 SQL INSERT：例如，在 JDBC 层面批处理几个 INSERT。由此，在一个工作单元期间，SQL 执行的发生会尽可能延后，而不是在你调用 `entityManager.persist(someItem)` 时发生。这只不过是排队插入操作以稍后执行，并且如果可能的话，指定标识符的值。但如果现在调用 `someItem.getId()`，则可能会得到 `null`，因为引擎未能在 INSERT 之前生成一个标识符。大体上，我们选用每次插入的生成策略，它会在 INSERT 之前独立生成标识符值。通常会选择一个共享且可并发访问的数据库序列。自增长列、列默认值或触发器生成的键仅在 INSERT 之后才可用。

我们还展示了每种标准 JPA 策略与其原生 Hibernate 对应项之间的关系。Hibernate 有机地发展，所以现在标准和原生策略之间有两组映射；我们在清单中称之为 Old 和 New。可以在你的 `persistence.xml` 文件中用 `hibernate.id.new_generator_mappings` 设置切换这一映射。默认是 `true`；也就是 New 映射。软件并非像酒那样越陈越香：

- **native**——自动选择其他策略，比如序列或标识，这取决于所配置的 SQL 方言。你必须查看在 `persistence.xml` 中配置的 SQL 方言的 Javadoc(或者源)。等效于使用 Old 映射的 `JPA GenerationType.AUTO`。
- **sequence**——使用一个名为 `HIBERNATE_SEQUENCE` 的原生数据库序列。该序列会在每次 INSERT 新行之前被调用。可以自定义该序列的名称并提供额外的 DDL 设置；参阅 `org.hibernate.id.SequenceGenerator` 类的 Javadoc。
- **sequence-identity**——通过在插入时调用一个数据库序列来生成键值：例如，`insert into ITEM(ID) values (HIBERNATE_SEQUENCE.nextval)`。该键值会在 INSERT 之后被检索，此行为与 `identity` 策略相同。支持与 `sequence` 策略相同的参数和属性类型；参阅 `org.hibernate.id.SequenceIdentityGenerator` 类及其父类的 Javadoc。
- **enhanced-sequence**——在得到支持的时候使用一个原生数据库序列；否则会退而借助一个具有单一列和行的额外数据库表，以模拟序列。其默认名称为 `HIBERNATE_SEQUENCE`。总是会在 INSERT 之前调用该数据库“序列”，无论 DBMS 是否支持真正的序列，都会提供相同的行为。支持 `org.hibernate.id.enhanced.Optimizer` 以避免在每次 INSERT 之前都访问数据库；默认不使用最优设置以及为每次 INSERT 获取一个新值。可以在第 20 章中找到更多的示例。对于所有的参数，都可以参阅 `org.hibernate.id.enhanced.SequenceStyleGenerator` 类的 Javadoc。等效于启用 New 映射的 `JPA GenerationType.SEQUENCE` 和 `GenerationType.AUTO`，很可能是内置策略的最佳选项。
- **seqhilo**——使用一个名为 `HIBERNATE_SEQUENCE` 的原生数据库序列，通过结合高/低位值优化 INSERT 之前的调用。如果从序列中检索到的高位值是 1，则接下来的 9 条插入的键值将是 11、12、13、.....、19。之后会再次调用该序列以获得下一个高位值(2 或更高的值)，且该过程会使用 21、22、23 等重复循环。可以使用 `max_lo` 参数配置最大的低位值(默认是 9)。遗憾的是，由于 Hibernate 代码中的一个奇怪之处，你不能在 `@GenericGenerator` 中配置这一策略。使用它的唯一方式是借助 `JPA GenerationType.SEQUENCE` 和 Old 映射。可以在一个(甚至可能是空的)

类上使用标准的 JPA `@SequenceGenerator` 注解来配置它。参阅 `org.hibernate.id.SequenceHiLoGenerator` 类及其父类的 Javadoc 以获得更多信息。可以考虑使用带有优化器的 `enhanced-sequence` 作为替代。

- **hilo**——使用一个名为 `HIBERNATE_UNIQUE_KEY` 的额外表以及与 `seqhilo` 策略相同的算法。该表具有单一列和行，保存序列的下一个值。默认最大低位值是 32 767，因此你最有可能希望使用 `max_lo` 参数来配置它。参阅 `org.hibernate.id.TableHiLoGenerator` 类的 Javadoc 以获取更多信息。我们不推荐这一传统策略；请使用带有优化器的 `enhanced-sequence` 作为替代。
- **enhanced-table**——使用一个名为 `HIBERNATE_SEQUENCES` 的额外表，它默认带有一个表示序列的行，且存储下一个值。当必须生成一个标识符值时，就会选择并更新这个值。可以配置这个生成器，转而使用多行：每行用于一个生成器；参阅 `org.hibernate.id.enhanced.TableGenerator` 的 Javadoc。等效于启用了 New 映射的 JPA `GenerationType.TABLE`。替换已过时但类似的 `org.hibernate.id.MultipleHiLoPerTableGenerator`，它是 JPA `GenerationType.TABLE` 的 Old 映射。
- **identity**——支持 DB2、MySQL、MS SQL Server 和 Sybase 中的 `IDENTITY` 和自增长列。用于主键列的标识符值将在 `INSERT` 一行时生成。它不具有可选项。遗憾的是，由于 Hibernate 代码中的一个奇怪之处，你不能在 `@GenericGenerator` 中配置这一策略。利用它的唯一方式是借助 JPA `GenerationType.IDENTITY` 和 Old 或 New 映射，让它成为 `GenerationType.IDENTITY` 的默认项。
- **increment**——在 Hibernate 启动时，会读取每个实体表的最大(数字)主键列值，并且在每次插入新行时递增该值。这在一个非聚集 Hibernate 应用程序具有对数据库的独占访问权时尤其有用；但不要在其他任何场景中使用它。
- **select**——Hibernate 不会生成一个键值，也不会 `INSERT` 语句中包含主键列。Hibernate 会预期 DBMS 在插入时为该列指定一个(模式默认或通过触发器生成的)值。之后，Hibernate 会在插入后使用一个 `SELECT` 查询检索该主键列。所需要的参数是 `key`，以指定用于 `SELECT` 的数据库标识符属性的名称(如 `id`)。这一策略并不非常有效，并且仅应该在具有不能直接返回所生成键的旧式 JDBC 驱动程序的情况下使用。
- **uuid2**——在应用层生成唯一的 128 位 UUID。当需要跨数据库的全局唯一标识符时就可以使用它(比如，每天晚上你要以批处理运行的方式从几个不同的生产数据库中将数据合并到一个存档中)。UUID 可以编码为 `java.lang.String`、`byte[16]` 或者实体类中的 `java.util.UUID` 属性。它可以替换遗留的 `uuid` 和 `uuid.hex` 策略。你要使用 `org.hibernate.id.UUIDGenerationStrategy` 来配置它；参阅 `org.hibernate.id.UUID-Generator` 类的 Javadoc 以了解更为详尽的信息。
- **guid**——利用 Oracle、Ingres、MS SQL Server 和 MySQL 上可用的 SQL 函数，以使用由数据库生成的全局唯一标识符。Hibernate 会在 `INSERT` 之前调用该数据库函数，它会映射到 `java.lang.String` 标识符属性。如果需要完全控制标识符的生成，可以使用实现了 `org.hibernate.id.IdentityGenerator` 接口的一个类的完全限定名称来配置 `@GenericGenerator` 的策略。

概括来说，我们推荐的标识符生成器策略如下：

- 一般而言，我们偏好使用插入前生成的策略，它会在 INSERT 之前独立生成标识符值。
- 使用 `enhanced-sequence`，它会在受支持的情况下使用原生数据库序列，否则会退而使用带有单个列和行的额外数据库表，以模拟序列。

我们假定从现在开始，你已经将标识符属性添加到域模型的实体类，并且在完成了每个实体及其标识符属性的基本映射之后，你要继续映射实体的值类型属性。我们将在第 5 章中介绍值类型映射。请继续阅读以便了解可简化和增强类映射的一些特殊选项。

4.3 实体映射选项

你现在已经用 `@Entity` 映射一个持久化类，所有其他设置都使用默认值，比如映射的 SQL 表名称。接下来一节将探究一些类级别的选项以及如何控制它们：

- 命名默认值和策略
- 动态 SQL 生成
- 实体可变性

这些是可选项；可以跳过这一节并在必须处理某个特定问题时再回过头来阅读。

4.3.1 控制名称

我们首先讨论实体类和表的命名。如果仅在可持久化的类上指定 `@Entity`，则默认映射的表名称会与类名称相同。注意我们是以大写字母编写 SQL 构件名称的，以便让它们更易于区分——SQL 实际上是不区分大小写的。因此，Java 实体类 `Item` 会映射到 `ITEM` 表。可以用 JPA `@Table` 注解重写这个表名称，如代码清单 4.3 所示。

代码清单 4.3 `@Table` 注解会重写所映射的表名称

路径：/model/src/main/java/org/jpwh/model/simple/User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {
    // ...
}
```

`User` 实体会映射到 `USER` 表；这是大多数 SQL DBMS 中的保留关键字。你不能使用具有该名称的表，因而要将它映射成 `USERS`。`@javax.persistence.Table` 注解也具有 `catalog` 和 `schema` 选项，如果数据库布局在命名前缀时需要的话，可以使用它们。

如果确实必须这样做，则可以加上引号来使用保留的 SQL 名称，甚至使用大小写敏感的名称。

1. 加引号的 SQL 标识符

总有些时候，尤其是在遗留的数据库中，你会碰到带有奇怪字符或空格的标识符，或


```

        return new Identifier("CE_" + name.getText(), name.isQuoted());
    }
}

```

`toPhysicalTableName()`这一重写的方法会将 `CE_` 添加到架构中所有生成的表名称之前。查看 `PhysicalNamingStrategy` 接口的 Javadoc；它提供了用于对列、序列和其他构件自定义命名的方法。

你必须在 `persistence.xml` 中启用该命名策略实现：

```

<persistence-unit>name="CaveatEmptorPU">
    ...
    <properties>
        <property name="hibernate.physical_naming_strategy"
            value="org.jpwh.shared.CENamingStrategy"/>
    </properties>
</persistence-unit>

```

用于命名自定义的第二个选项是 `ImplicitNamingStrategy`。尽管物理命名策略是在最低的层面发挥作用，但是在架构的构件名称最终产生之前，会先调用隐式命名策略。如果映射一个实体类，并且不对一个显式名称使用 `@Table` 注解，则会询问该隐式命名策略实现，表名称应该是什么。这是基于像实体名称和类名称这样的因素的。`Hibernate` 附带了几种策略来实现遗留或兼容 JPA 的默认名称。默认策略是 `ImplicitNamingStrategyJpaCompliantImpl`。

我们来快速介绍另一个相关问题：命名用于查询的实体。

2. 命名用于查询的实体

默认情况下，所有的实体名称都是自动导入到查询引擎的命名空间中的。换句话说，可以方便地在 JPA 查询字符串中使用不带有包前缀的简易类名称：

```

List result = em.createQuery("select i from Item i")
    .getResultList();

```

这仅在你的持久化单元中有一个 `Item` 类时有用。如果在另一个包中添加另一个 `Item` 类，则应该对其中一个类重命名以便用于 JPA，前提是希望继续在查询中使用简易格式的话：

```

package my.other.model;
@javax.persistence.Entity(name = "AuctionItem")
public class Item {
    // ...
}

```

简易查询格式现在会为 `my.other.model` 包中的 `Item` 类 `select i from AuctionItem i`。因此你要使用另一个包中的另一个 `Item` 类来解决该命名冲突。当然，可以一直使用带有包前缀的完全限定的长名称。

到此我们就完成了 `Hibernate` 中命名选项的介绍。接下来，我们将探讨 `Hibernate` 如何生成包含这些名称的 SQL。

Hibernate 特性

4.3.2 动态 SQL 生成

默认情况下,在启动期间,Hibernate 会在创建持久化单元时为每一个持久化类创建 SQL 语句。这些语句都是用于读取单行、删除一行等的创建、读取、更新和删除(CRUD)操作。预先在内存中存储这些操作,系统开销会更低,而非在运行时每次必须执行如此简单的查询时生成 SQL 字符串。此外,如果只有很少的语句的话,那么在 JDBC 级别准备好的语句缓存会更加高效。

Hibernate 如何在启动时创建 UPDATE 语句?毕竟,此时要更新的列还是未知的。这个问题的答案是,所生成的 SQL 语句会更新所有的列,并且如果特定列的值未修改的话,则该语句会将它设置为旧值。

在某些情况下,比如对于具有数百个列的遗留表,即便是最简单的操作(例如只需要更新一列),其 SQL 语句也会很大,这时你应该禁用这一启动 SQL 生成,并且切换到运行时的动态语句生成。非常大的实体数量也会影响启动时间,因为 Hibernate 必须预先为 CRUD 生成所有的 SQL 语句。如果必须为数千个实体缓存一系列语句的话,那么这一查询语句缓存的内存消耗也将很高。这在具有有限内存的虚拟化环境或者低效设备中会是一个问题。

要在启动时禁用 INSERT 和 UPDATE SQL 语句的生成,需要使用原生的 Hibernate 注解:

```
@Entity
@org.hibernate.annotations.DynamicInsert
@org.hibernate.annotations.DynamicUpdate
public class Item {
    // ...
}
```

通过启用动态插入和更新,就可告知 Hibernate 在需要时(而非预先)生成 SQL 字符串。UPDATE 将仅包含具有更新后的值的列,而 INSERT 将仅包含非空列。

我们将在第 17 章中再次探讨 SQL 生成和自定义 SQL。如果实体不可变的话,有时候完全可以避免生成 UPDATE 语句。

4.3.3 让实体不可变

一个特定类的实例可以是不可变的。例如,在 CaveatEmptor 中,用于一件商品的 Bid 是不可变的。因此, Hibernate 永远不需要对 BID 表执行 UPDATE 语句。Hibernate 还可以进行其他一些优化,例如当映射下一个示例中所示的不可变类时避免脏检查。此处,该 Bid 类是不可变的,并且实例永远不会被修改:

```
@Entity
@org.hibernate.annotations.Immutable
public class Bid {
    // ...
}
```


如果公开了用于类的所有属性的非公共设置方法，则 POJO 就是不可变的——所有的值都是在构造函数中设置的。Hibernate 应该会在加载和存储实例时直接访问字段。我们在这一章前面的内容中谈到过这一点：如果 `@Id` 注解位于一个字段上，则 Hibernate 会直接访问字段，并且可以按照你认为合适的方式随意设计获取和设置方法。此外，要记住并非所有的框架都适用不带有设置方法的 POJO；例如，JSF 不会直接访问字段来填充一个实例。

当不能在你的数据库架构中创建一个视图时，可以将不可变的实体类映射到一个 SQL SELECT 查询。

4.3.4 将一个实体映射到子查询

有时候你的 DBA 不允许你修改数据库架构；即使是添加一个新的视图可能都不行。我们假设你希望创建包含一个拍卖 Item 标识符和对该商品的出价次数的视图。

使用 Hibernate 注解，可以创建一个应用程序级别的视图，以及一个映射到 SQL SELECT 的只读实体类：

路径：/model/src/main/java/org/jpwh/model/advanced/ItemBidSummary.java

```
@Entity
@org.hibernate.annotations.Immutable
@org.hibernate.annotations.Subselect(
    value = "select i.ID as ITEMID, i.ITEM_NAME as NAME, " +
            "count(b.ID) as NUMBEROFBIDS " +
            "from ITEM i left outer join BID b on i.ID = b.ITEM_ID " +
            "group by i.ID, i.ITEM_NAME"
)
@org.hibernate.annotations.Synchronize({"Item", "Bid"})
public class ItemBidSummary {

    @Id
    protected Long itemId;

    protected String name;

    protected long numberOfBids;

    public ItemBidSummary() {
    }

    // Getter methods...
    // ...
}
```

待解决问题：表名称是区分大小写的(Hibernate 漏洞 HHH- 8430)

在加载了 ItemBidSummary 的一个实例时，Hibernate 会将你的自定义 SQL SELECT 作为一个子查询来执行：

路径：/examples/src/test/java/org/jpwh/test/advanced/MappedSubselect.java

```
ItemBidSummary itemBidSummary = em.find(ItemBidSummary.class, ITEM_ID);
// select * from (
//     select i.ID as ITEMID, i.ITEM_NAME as NAME, ...
// ) where ITEMID = ?
```

你应该在`@org.hibernate.annotations.Synchronize`注解中列出你的 SELECT 中所有引用过的表名称(在编写本书时, Hibernate 还有一个在问题 HHH-8430【见注 4.1】下跟踪的漏洞, 它会让所同步的表名称区分大小写)。之后 Hibernate 就会获悉它必须在执行对 `ItemBidSummary` 的查询之前刷新 `Item` 和 `Bid` 实例的修改:

路径: `/examples/src/test/java/org/jpwh/test/advanced/MappedSubselect.java`

```
Item item = em.find(Item.class, ITEM_ID);
item.setName("New name");

// ItemBidSummary itemBidSummary = em.find(ItemBidSummary.class, ITEM_ID);

Query query = em.createQuery(
    "select ibs from ItemBidSummary ibs where ibs.itemId = :id"
);
ItemBidSummary itemBidSummary =
    (ItemBidSummary)query.setParameter("id", ITEM_ID).getSingleResult();
```

在由标识符检索之前不会刷新

如果所同步的表受到影响, 则在查询之前自动刷新

注意, Hibernate 不会在 `find()` 操作之前自动刷新——如果必要的话, 仅会在执行一个 Query 之前自动刷新。Hibernate 可以检测该修改的 `Item` 会否影响查询结果, 因为 `ITEM` 表是与 `ItemBidSummary` 同步的。因此, 对 `ITEM` 行的刷新和 `UPDATE` 是必需的, 以避免查询返回过时的数据。

4.4 本章小结

- 实体是系统中粗粒度的类。它们的实例具有独立的生命周期及其自己的标识, 而且许多其他实例可以引用它们。
- 另一方面, 值类型依赖于特定的实体类。值类型实例会绑定到其所属的实体实例, 并且只有一个实体实例能够引用它——它不具有单独的标识。
- 我们探讨了 Java 表示、对象相等性以及数据库标识, 以及如何生成优秀的主键。你了解了 Hibernate 提供了哪些开箱即用的主键值生成器, 以及如何使用和扩展这一标识符系统。
- 我们探讨了一些有用的类映射选项, 比如命名策略和动态 SQL 生成。

5.1 映射基本属性

映射持久化类的时候, 无论它是实体类还是可嵌入类型(5.2 节将进一步介绍), 其所有的属性和默认值, 都是其持久化的。对于持久化类的属性的默认 JPA 规则如下:

本章内容简介：

- 映射基本属性
- 映射可嵌入组件
- 控制 Java 和 SQL 类型之间的映射

在第 4 章专门介绍实体以及相应类和标识映射选项之后，本章将着重介绍各种格式的值类型。我们将值类型划分成两类：JDK 自带的基本值类型类，如 `String`、`Date`、基元及其包装器；开发人员定义的值类型类，如 `CaveatEmptor` 中的 `Address` 和 `MonetaryAmount`。

本章首先使用 JDK 类型映射持久化属性并学习基本映射注解。会看到如何处理属性的各个方面：重写默认设置、自定义访问以及生成的值。还会看到如何将 SQL 用于派生属性和转换后的列值。使用时序属性和映射枚举包装基本属性。然后探讨自定义值类型类以及将它们映射成可嵌入的组件。还将学习类如何与数据库架构相关联以及如何让你的类可嵌入，同时如何允许重写嵌入式特性。我们将通过映射嵌套组件来完成对可嵌入组件的介绍。最后，会探讨如何使用灵活的 JPA 转换器在较低的级别自定义属性值的加载和存储，该转换器是每个 JPA 提供程序的标准扩展点。

JPA 2 中主要的新功能

- 使用 `@Access` 注解，通过用于实体层次结构或独立属性的字段或者属性 `getter/setter` 方法进行可切换访问
- 嵌入式组件类的多层嵌套，以及使用圆点标记法将 `@AttributeOverride` 应用到嵌套的嵌入式属性的能力
- 增加了用于基本类型属性的 `Converter` API，因而可以控制如何加载和存储值以及在必要时转换它们。

5.1 映射基本属性

映射持久化类的时候，无论它是实体还是可嵌入类型(5.2 节将进一步介绍)，其所有的属性都会默认认为其是持久的。用于持久化类的属性的默认 JPA 规则如下：

- 如果属性是基元或基元包装器，或者是 `String`、`BigInteger`、`BigDecimal`、`java.util.Date`、`java.util.Calendar`、`java.sql.Date`、`java.sql.Time`、`java.sql.Timestamp`、`byte[]`、`Byte[]`、`char[]` 或 `Character[]` 类型，则它会被自动持久化。Hibernate 会在具有合适的 SQL 类型且名称与属性名称相同的列中加载和存储该属性的值。
- 如果将属性的类注解为 `@Embeddable`，或者将属性本身映射为 `@Embedded`，则 Hibernate 会将该属性映射为所属类的可嵌入组件。本章稍后将用 `CaveatEmptor` 的 `Address` 和 `MonetaryAmount` 这两个可嵌入类来探讨组件的嵌入。
- 如果属性的类型是 `java.io.Serializable`，则它的值会以其序列化格式存储。这通常并非你所期望的，并且你应该总是映射 Java 类而非在数据库中存储一堆字节。想象一下几年后应用程序停用时使用这个二进制信息维护数据库会多困难吧。
- Hibernate 会在启动时抛出一个异常，提示它不理解该属性的类型。

按异常进行配置的这个方法意味着你不必注解一个属性以便让其持久化；只需要配置特殊情况中的映射即可。JPA 中提供了几个注解来自定义和控制基本属性映射。

5.1.1 重写基本属性的默认设置

可能不希望持久化一个实体类的所有属性。例如，尽管使用持久化 `Item#initialPrice` 属性是合理的，但如果仅在运行时计算和使用它的值，则不应该持久化 `Item#totalPriceIncludingTax` 属性，因此不应该在数据库中存储它。要排除一个属性，需要使用 `@javax.persistence.Transient` 注解或者使用 `Javatransient` 关键字标记该字段或者该属性的 `getter` 方法。`transient` 关键字通常仅排除用于 Java 序列化的字段，但它也可以由 JPA 提供程序识别。

马上会回过头来介绍字段或 `getter` 方法上注解的放置位置。像之前那样假定，由于已经在字段上放置了 `@Id`，Hibernate 会直接访问字段。因此，所有其他 JPA 和 Hibernate 映射注解也位于字段上。

如果不希望依赖属性映射默认设置，那么可以将 `@Basic` 注解应用到某个特定属性，如 `Item` 的 `initialPrice`：

```
@Basic(optional = false)
BigDecimal initialPrice;
```

必须承认，这个注解没太大用处。它只有两个参数。一个是此处显示的 `optional`，它会将属性在 Java 对象级别标记为不可选。默认情况下，所有的持久化属性都是可为空并且可选的；`Item` 可能有一个未知的 `initialPrice`。如果 SQL 架构中的 `INITIALPRICE` 列上有 `NOT NULL` 约束，那么将 `initialPrice` 属性映射为不可选是合理的。如果 Hibernate 正在生成 SQL 架构，则它会为不可选属性自动包含 `NOT NULL` 约束。

现在，当存储 `Item` 并且忘记在 `initialPrice` 字段上设置一个值的时候，Hibernate 会在使用 SQL 语句访问数据库之前抛出一个异常。Hibernate 知道需要一个值来执行 `INSERT` 或 `UPDATE`。如果不将属性标记为可选并且试着保存 `NULL`，数据库会拒绝执行该 SQL 语句，

而 Hibernate 将抛出一个违反约束的异常。最终的结果并没有太多差别，但避免使用会失败的语句访问数据库这一做法会更为简洁。在 12.1 节探究优化策略时讨论 @Basic 的另一个参数，fetch 选项。

相较于 @Basic，大多数工程师都会使用更为通用的 @Column 注解来声明可为空属性：

```
@Column(nullable = false)
BigDecimal initialPrice;
```

现在已经介绍了声明是否需要属性值的 3 种方式：使用 @Basic 注解、@Column 注解以及稍早前在 3.3.2 小节中介绍的 Bean Validation @NotNull 注解。这 3 种方式对于 JPA 提供程序都有相同的效果：在数据库架构中保存和生成 NOT NULL 约束时，Hibernate 会进行 null 检查。推荐使用 BeanValidation @NotNull 注解，以便能够手动验证 Item 实例和/或让表示层中的用户接口代码自动执行验证检查。

@Column 注解也可以重写属性名称到数据库列的映射：

```
@Column(name = "START_PRICE", nullable = false)
BigDecimal initialPrice;
```

@Column 注解有一些其他的参数，大多数都是控制 SQL 级别的详细信息的，如目录 (catalog) 和架构 (schema) 名称。由于很少会用到它们，因而本书中只在必要时对其进行介绍。属性注解并非总是位于字段上，而且可能不希望 Hibernate 直接访问字段。

5.1.2 自定义属性访问

持久化引擎会直接通过字段访问类的属性，或者间接通过 getter 和 setter 方法进行访问。一个已注解过的实体会从强制的 @Id 注解位置继承默认设置。例如，如果已经在一个字段而非 getter 方法上声明了 @Id，则该实体的所有其他映射注解预期都会在字段上。注解绝不会位于 setter 方法上。

默认访问策略并非仅适用于单个实体类。任何 @Embedded 类都会继承默认设置或者其所属根实体类明确声明过的访问策略。本章稍后将介绍嵌入式组件。此外，Hibernate 会使用默认设置或所映射的实体类的明确声明过的访问策略来访问所有 @MappedSuperclass 属性。继承性是第 6 章的主题。

JPA 规范提供了用于重写默认行为的 @Access 注解，这要用到参数 AccessType.FIELD 和 AccessType.PROPERTY。如果在类/实体级别设置 @Access，则 Hibernate 会根据所选择策略访问类的所有属性。然后要设置所有其他映射注解(其中包括 @Id)，分别在字段或 getter 方法上设置。

还可以使用 @Access 注解重写单个属性的访问策略。下面用一个示例来探究这一点，见代码清单 5.1。

代码清单 5.1 重写用于 name 属性的访问策略

路径：/model/src/main/java/org/jpwh/model/advanced/Item.java

```
@Entity
public class Item {
```

```

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;
    @Access (AccessType.PROPERTY)
    @Column(name = "ITEM_NAME")
    protected String name;

    public String getName() {
        return name;
    }

```

映射仍在期望的这里

① @Id 位于字段上

② 将属性切换成运行时访问

③ 在加载/存储时调用

- ① Item 实体默认为字段访问。@Id 位于字段上(还要将脆弱的 ID_GENERATOR 字符串移动到一个常量中)。
- ② name 字段上的 @Access(AccessType.PROPERTY) 设置会将这一特定属性切换成由 JPA 提供程序通过 getter/setter 方法在运行时访问。
- ③ 在加载和存储商品时, Hibernate 会调用 getName() 和 setName()。

注意, 像 @Column 这样的其他映射注解的位置不会改变——只有运行时对实例的访问方式会改变。

现在调换一下顺序: 如果实体的默认(或显式)访问类型是通过属性 getter 和 setter 方法实现, 则 getter 方法上的 @Access(AccessType.FIELD) 将告知 Hibernate 直接访问该字段。所有其他的映射信息仍将必须位于 getter 方法上, 而非位于字段上。

Hibernate 特性

Hibernate 有一个很少用到的扩展: noop 属性访问器。这听上去可能很奇怪, 但它能让你在查询中引用虚属性。如果有一个只想在 JPA 查询中使用的数据库列, 那么这会很有用。例如, 假设 ITEM 数据库表有一个 VALIDATED 列, 并且 Hibernate 应用程序不会通过域模型访问这一列。它可能是一个遗留的列或者由另一个应用程序或数据库触发器维护的列。你想要的就是在 JPA 查询中引用该列, 如 `select i from Item i where i.validated = true` 或者 `select i.id, i.validated from Item i`。域模型中的 Java Item 类没有这个属性; 因此没有地方可以放置注解。映射这样一个虚属性的唯一方法是使用 hbm.xml 原生元数据文件:

```

<hibernate-mapping>
  <class name="Item">
    <id name="id">
      ...
    </id>
    <property name="validated"
      column="VALIDATED"
      access="noop"/>
  </class>
</hibernate-mapping>

```

这个映射会告知 Hibernate 希望访问 Item#validated 虚属性, 它是在查询中映射到 VALIDATED 列的; 但对于运行时的值读取/写入来说, 希望在 Item 的实例上“不做任何操

作”。类不具有该特性。记住这样的原生映射文件必须完整：Item 类上的任何注解现在都会被忽略！

如果没有合适的内置访问策略，那么可以通过实现 `org.hibernate.property.PropertyAccessor` 接口来定义自己的自定义属性访问策略。通过在 Hibernate 扩展注解中设置其完全限定名称就可以启用自定义访问器：`@org.hibernate.annotations.AttributeAccessor("my.custom.Accessor")`。注意，`AttributeAccessor` 是 Hibernate 4.3 中的新功能，它替代了过时的、且容易与 JPA 枚举 `javax.persistence.AccessType` 混淆的 `org.hibernate.annotations.AccessType`。

有些属性不会映射到一个列。尤其是，派生属性会使用其来自 SQL 表达式的值。

5.1.3 使用派生属性

派生属性的值是在运行时通过估算由 `@org.hibernate.annotations.Formula` 注解声明的 SQL 表达式来计算的；参见代码清单 5.2。

代码清单 5.2 两个只读的派生属性

```
@org.hibernate.annotations.Formula(  
    "substr(DESCRIPTION, 1, 12) || '...'"  
)  
protected String shortDescription;  
  
@org.hibernate.annotations.Formula(  
    "(select avg(b.AMOUNT) from BID b where b.ITEM_ID = ID)"  
)  
protected BigDecimal averageBidAmount;
```

指定的 SQL 公式会在每次从数据库检索 Item 实体时估算，其他任何时候都不估算，所以如果其他属性被修改了，则结果可能会过期。这些属性绝不会出现在 SQL 的 INSERT 或 UPDATE 语句中，只会出现在 SELECT 语句中。估算是发生在数据库中的；在加载实例时，Hibernate 会在 SELECT 子句中嵌入 SQL 公式。

公式可以引用数据库表的列，它们可以调用 SQL 函数，甚至可以包含 SQL 子查询。在代码清单 5.2 中，调用了 `SUBSTR()` 函数，以及 `||` 连接操作符。该 SQL 表达式会被原样传递到底层数据库；如果不想费心，可以借助于供应商专有的操作符或关键字，并且将映射元数据绑定到一个特定的数据库产品。注意，不合格的列名称会引用派生属性所归属的类表的列。

数据库只会在 Hibernate 从数据库检索实体实例时估算公式中的 SQL 表达式。Hibernate 还支持名为列转换器的公式变体，以允许编写用于读取和写入属性值的自定义 SQL 表达式。

5.1.4 转换列值

假设有一个名为 `IMPERIALWEIGHT` 的数据库列，它会以磅为单位存储 Item 的重量。不过，应用程序有一个以千克为单位的 `Item#metricWeight` 属性，所以从 Item 表读取以及向该表写入行时必须转换该数据库列的值。可以使用一个 Hibernate 扩展来实现该转换：

@org.hibernate.annotations.ColumnTransformer 注解，见代码清单 5.3。

代码清单5.3 使用SQL表达式转换列值

```
@Column(name = "IMPERIALWEIGHT")
@org.hibernate.annotations.ColumnTransformer(
    read = "IMPERIALWEIGHT / 2.20462",
    write = "? * 2.20462"
)
protected double metricWeight;
```

在从 ITEM 表读取行时，Hibernate 会嵌入表达式 IMPERIALWEIGHT / 2.20462，所以该计算发生在数据库中并且 Hibernate 会将结果中的测量值返回给应用层。对于该列的写入，Hibernate 会在强制的单个占位符(问号)上设置测量值，而你的 SQL 表达式会计算要插入或更新的实际值。

Hibernate 还会在查询约束中应用列转换器。例如，以下查询会检索所有两千克重的商品：

```
List<Item> result =
    em.createQuery("select i from Item i where i.metricWeight = :w")
        .setParameter("w", 2.0)
        .getResultList();
```

此查询由 Hibernate 执行的实际 SQL 在 WHERE 子句中包含了以下约束：

```
// ...
where
    i.IMPERIALWEIGHT / 2.20462=?
```

注意，数据库很可能不能够依赖这一约束的索引；你将看到完整的表扫描，因为必须计算，所有 ITEM 行的重量，以估算该约束。

另一种特殊种类的属性依赖于数据库生成的值。

5.1.5 生成的以及默认的属性值

数据库有时候会生成属性值，通常是在首次插入行时。数据库生成值的例子包括创建时间戳、商品的默认价格以及为每次修改所运行的触发器。

通常，Hibernate 应用程序需要在保存之后刷新实例，这些实例包含数据库为之生成值的任何属性。这意味着可能必须对数据库进行另一次交互，以便在插入和更新行之后读取值。将属性标记为被生成，让应用程序可以将这一职责委托给 Hibernate。实质上，无论何时 Hibernate 为已经声明了生成属性的实体执行 SQL INSERT 或 UPDATE，随后都会立即执行 SELECT，以检索生成的值。

使用@org.hibernate.annotations.Generated 注解标记生成属性，见代码清单 5.4。

代码清单5.4 数据库生成的属性值

```
@Temporal(TemporalType.TIMESTAMP)
@Column(insertable = false, updatable = false)
@org.hibernate.annotations.Generated(
```

```

    org.hibernate.annotations.GenerationTime.ALWAYS
)
protected Date lastModified;
@Column(insertable = false)
@org.hibernate.annotations.ColumnDefault("1.00")
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
protected BigDecimal initialPrice;

```

GenerationTime 的可用设置是 ALWAYS 和 INSERT。

使用 ALWAYS, Hibernate 会在每次执行 SQL 的 UPDATE 或 INSERT 语句之后刷新实体实例。示例假定数据库触发器会保存当前的 lastModified 属性。还应该使用 @Column 的 updatable 和 insertable 参数将该属性标记为只读。如果这两者都设置为 false, 则该属性的列将永远不会出现在 INSERT 或 UPDATE 语句中, 并且要让数据库生成其值。

使用 GenerationTime.INSERT, 刷新将只发生在 SQL INSERT 语句之后, 以检索由数据库提供的默认值。Hibernate 还会将该属性映射为非 insertable。@ColumnDefault Hibernate 注解会在 Hibernate 导出和生成 SQL 架构 DDL 时设置列的默认值。

时间戳常常是自动生成的值, 要么像前一个示例(代码清单 5.4)一样由数据库生成, 要么由应用程序生成。来更详细地查看一下代码清单 5.4 中的 @Temporal 注解。

5.1.6 时序属性

代码清单 5.4 中的 lastModified 属性是 java.util.Date 类型, 并且 SQL INSERT 上的数据库触发器生成了它的值。JPA 规范要求使用 @Temporal 注解时序属性, 以声明所映射列的准确 SQL 数据库类型。Java 时序类型是 java.util.Date、java.util.Calendar、java.sql.Date、java.sql.Time 和 java.sql.Timestamp。Hibernate 也支持 JDK 8 中的 java.time 包的类(实际上, 如果一个转换器被应用于(或适合于)属性, 则不需要注解。本章稍后将再次介绍转换器)。

代码清单 5.5 显示了兼容 JPA 的示例: 典型的时间戳“这个商品被创建在”保存一次但永远不会更新。

代码清单 5.5 必须使用 @Temporal 注解的时序类型的属性

```

@Temporal(TemporalType.TIMESTAMP)
@Column(updatable = false)
@org.hibernate.annotations.CreationTimestamp
protected Date createdOn;

// Java 8 API
// protected Instant reviewedOn;

```

JPA 说明 @Temporal 是需要的, 但 Hibernate 默认使用没有它的 TIMESTAMP。

可用的 TemporalType 选项是 DATE、TIME 和 TIMESTAMP, 以确定该时序值的哪部分应该存储在数据库中。

Hibernate 特性

当未提供@Temporal 注解时,Hibernate 会默认使用 TemporalType.TIMESTAMP。此外,已经使用过 Hibernate 注解@CreationTimestamp 来标记该属性。这是 5.1.5 小节中@Generated 注解的同级注解:它会告知 Hibernate 自动生成属性值。在这种情况下,Hibernate 会在它将实体实例插入到数据库之前将值设置为当前时间。类似的一个内置注解是@UpdateTimestamp。也可以编写和配置自定义的值生成器,在应用程序或数据库中运行。可以查看一下 org.hibernate.annotations.GeneratorType 和 ValueGenerationType。

另一个特殊属性类型是枚举类型。

5.1.7 映射枚举

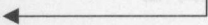
枚举类型是一个常用的 Java 习惯用语,它指的是类具有恒定(较少)数量的不可变实例。例如,在 CaveatEmptor 中,可以将其应用到拍卖:

```
public enum AuctionType {  
    HIGHEST_BID,  
    LOWEST_BID,  
    FIXED_PRICE  
}
```

现在可以在每个 Item 上设置合适的 auctionType:

```
@NotNull  
@Enumerated(EnumType.STRING)  
protected AuctionType auctionType = AuctionType.HIGHEST_BID;
```

默认使用 ORDINAL



不使用@Enumerated 注解,则 Hibernate 会存储值的 ORDINAL 位置。也就是说,它会为 HIGHEST_BID 存储 1、为 LOWEST_BID 存储 2、而为 FIXED_PRICE 存储 3。这是一个脆弱的默认设置;如果对 AuctionType 枚举进行变更,则现有的值可能就不再映射到相同位置。因此 EnumType.STRING 选项是一个更好的选择;Hibernate 会按照原样存储枚举值的标签。

到此就完成了对基本属性及其映射选项的介绍。到目前为止,已经介绍了 JDK 提供的类型的属性,如 String、Date 和 BigDecimal。域模型还具有自定义值类型的类,就是在 UML 图中具有组合关联的那些。

5.2 映射可嵌入组件

到目前为止,所映射的域模型的类都是实体类,每个类都有其自己的生命周期和标识。不过,User 类与 Address 类有一个特殊类型的关联,如图 5-1 所示。



图 5-1 User 和 Address 的组合

在对象建模术语中，这一关联就是一类聚合——关系的一部分。聚合是关联的一种强形式；它具有关于对象生命周期的一些附加语义。在这个例子中，有一个甚至更强的形式——组合，其中组成部分的生命周期完全依赖于整体的生命周期。像 Address 这样的 UML 中的组合类通常是用于对象/关系映射的候选值类型。

5.2.1 数据库架构

下面映射这样一种组合关系：Address 作为值类型，具有与 String 或 BigDecimal 相同的语义，User 作为一个实体。首先，图 5-2 中所示为要构建的 SQL 架构。

<< Table >> USERS	
ID <<PK>>	组件列
USERNAME	
FIRSTNAME	
LASTNAME	
STREET	
ZIPCODE	
CITY	
BILLING_STREET	
BILLING_ZIPCODE	
BILLING_CITY	

图 5-2 内嵌在实体表中的组件的列

这里只有一个映射表 USERS 用于 User 实体。这个表嵌入了组件的所有详情，其中单个行会保存特定 User 及其 homeAddress 和 billingAddress。如果另一个实体具有对 Address 的引用——如 Shipment#deliveryAddress——那么 SHIPMENT 表也要有存储 Address 所需要的所有列。

此架构反映出了值类型语义：特定的 Address 不能被共享；它没有自己的标识。其主键是它所属实体拥有映射的数据库标识符。内嵌的组件有独立的生命周期：在保存所拥有的实体实例时，也会保存该组件实例。当其所拥有实体实例被删除时，也会删除该组件实例。Hibernate 甚至不必为此执行任何特殊 SQL；所有数据都位于一行中。

使用“比表多的类”是 Hibernate 支持细粒度域模型的方式。下面编写用于这一结构的类和映射。

5.2.2 让类可嵌入

Java 没有组合的概念——类或属性不能被标记为组件或组合生命周期。与实体的唯一区别是数据库标识符：组件类没有独立的标识；因此，组件类不需要标识符属性或标识符映射。它是一个简单 POJO，可以在代码清单 5.6 中看到这一点。

代码清单5.6 Address类：一个可嵌入组件

路径：/model/src/main/java/org/jpwh/model/simple/Address.java

@Embeddable

```
public class Address {
```

@NotNull ← 忽略了 DDL 生成

@Column(nullable = false)

protected String street;

@NotNull

@Column(nullable = false, length = 5) ← 重写 VARCHAR(255)

protected String zipcode;

@NotNull

@Column(nullable = false)

protected String city;

protected Address() {

}

public Address(String street, String zipcode, String city) {

 this.street = street;

 this.zipcode = zipcode;

 this.city = city;

}

public String getStreet() {

 return street;

}

public void setStreet(String street) {

 this.street = street;

}

public String getZipcode() {

 return zipcode;

}

public void setZipcode(String zipcode) {

 this.zipcode = zipcode;

}

public String getCity() {

 return city;

}

public void setCity(String city) {

 this.city = city;

}

}

① 使用 @Embeddable 而非 @Entity

用于 DDL 生成

② 无参构造函数

③ 便利性构造函数

① 相较于 @Entity，这个组件 POJO 是使用 @Embeddable 标记的。它没有标识符属性。

② Hibernate 会调用这个无参构造函数来创建一个实例，然后直接填充字段。

③ 为了方便，可以使用额外的(公共)构造函数。

可嵌入类的属性默认都是持久化的，就像一个持久化实体类的属性。可以使用相同的注解配置属性映射，如 `@Column` 或 `@Basic`。Address 类的属性映射到 STREET、ZIPCODE 和 CITY 列，并且使用了 NOT NULL 约束。

问题：Hibernate 验证器不会生成 NOT NULL 约束

在编写本书时，使用 Hibernate 验证器仍然有一个开放性问题：在生成数据库架构时，Hibernate 不会将可嵌入组件属性上的 `@NotNull` 约束映射到 NOT NULL 约束。为了 Bean 验证，Hibernate 只会在运行时的组件属性上使用 `@NotNull`。必须使用 `@Column(nullable = false)` 映射属性，以生成架构中的约束。Hibernate 问题数据库将此问题作为 HVAL-3 进行跟踪。

这就是完整的映射。没有与 User 实体有关的特殊内容：

路径：/model/src/main/java/org/jpwh/model/simple/User.java

`@Entity`

`@Table(name = "USERS")`

```
public class User implements Serializable {
```

```
    @Id
```

```
    @GeneratedValue(generator = Constants.ID_GENERATOR)
```

```
    protected Long id;
```

```
    public Long getId() {
```

```
        return id;
```

```
    }
```

```
    protected Address homeAddress;
```

Address 是 `@Embeddable`；这里不需要注解。

```
    public Address getHomeAddress() {
```

```
        return homeAddress;
```

```
    }
```

```
    public void setHomeAddress(Address homeAddress) {
```

```
        this.homeAddress = homeAddress;
```

```
    }
```

```
    // ...
```

```
}
```

Hibernate 会检测 Address 类是否使用 `@Embeddable` 进行了注解；STREET、ZIPCODE 和 CITY 列是在 USERS 表上映射的，即所属实体的表。

本章稍早前探讨属性访问时提到过，可嵌入组件会继承其所属实体的访问策略。这意味着 Hibernate 将使用与用于 User 属性相同的策略来访问 Address 类的属性。这一继承性也会影响可嵌入组件类中映射注解的位置。其规则如下：

- 如果嵌入组件所属的 `@Entity` 是使用字段访问来映射的，隐式使用字段上的 `@Id` 或者显式使用类上的 `@Access(AccessType.FIELD)`，那么该嵌入组件类的所有映射注解都预期会位于该组件类的字段上。Hibernate 会期待 Address 字段上的注解并且直接在运行时读取/写入字段。Address 上的 getter 和 setter 方法是可选的。

- 如果嵌入组件所属的@Entity 是使用属性访问映射的,隐式使用 getter 方法上的@Id 或者显式使用类上的@Access(AccessType.PROPERTY),那么该嵌入组件类的所有映射注解都预期会位于该组件类的 getter 方法上。之后 Hibernate 会通过调用可嵌入组件类上的 getter 和 setter 方法来读取和写入值。
- 如果所属实体类的可嵌入属性——上一个示例中的 User#homeAddress——是使用 @Access(AccessType.FIELD)标记的,那么 Hibernate 会预期注解位于 Address 类的字段上并且在运行时访问字段。
- 如果所属实体类的可嵌入属性——上一个示例中的 User#homeAddress——是使用 @Access(AccessType.PROPERTY)标记的,那么 Hibernate 会预期注解位于 Address 类的 getter 方法上并且在运行时访问 getter 和 setter 方法。
- 如果@Access 注解了可嵌入类本身,那么 Hibernate 会将所选择的策略用于读取该可嵌入类上的映射注解并且在运行时访问。

Hibernate 属性

还有一个警告需要记住:没有完善的方式来表示 Address 的 null 引用。如果 STREET、ZIPCODE 和 CITY 列可为空,考虑一下会发生什么吧。当 Hibernate 加载不带有任何地址信息的 User 时, someUser.getHomeAddress()应该返回什么呢? 这个例子中 Hibernate 会返回 null。Hibernate 还会在组件所有的映射列中将 null 嵌入属性存储为 NULL 值。因而,如果存储一个带有“空”Address 的 User(有一个 Address 实例但其所有的属性都是 null),则在加载该 User 时不会返回任何 Address 实例。这会是反常的;另一方面,无论如何可能都不应使用可为空的列并且避免三元逻辑。

你应该重写 Address 的 equals()和 hashCode()方法,并且通过值来对比实例。只要你不必对比实例,那么这就不是至关重要的:例如,通过将实例放入 HashSet 来对比。我们将在稍后的集合上下文中探讨这个问题;参阅 7.2.1 节。

在更为现实的场景中,一个用户可能会将不同的地址用于不同的目的。图 5-1 显示了 User 和 Address 之间额外的组合关系: billingAddress。

5.2.3 重写嵌入属性

billingAddress 是 User 类的另一个嵌入组件属性,所以必须在 USERS 表中保存另一个 Address。这就造成了映射冲突:到目前为止,架构中只有在 STREET、ZIPCODE 和 CITY 中存储一个 Address 的列。

需要额外的列来存储用于每个 USERS 行的另一个 Address。映射 billingAddress 时,要重写列名称:

路径: /model/src/main/java/org/jpwh/model/simple/User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {
```

@Embedded

← 非必要

```
@AttributeOverrides({
    @AttributeOverride(name = "street",
        可为 NULL! —————> column = @Column(name = "BILLING_STREET")),
    @AttributeOverride(name = "zipcode",
        column = @Column(name = "BILLING_ZIPCODE", length = 5)),
    @AttributeOverride(name = "city",
        column = @Column(name = "BILLING_CITY"))
})
protected Address billingAddress;

public Address getBillingAddress() {
    return billingAddress;
}
public void setBillingAddress(Address billingAddress) {
    this.billingAddress = billingAddress;
}
// ...
}
```

`@Embedded` 注解实际上不是必要的。它是 `@Embeddable` 的一个替代注解：标记所属实体类中的组件类或属性(这两者都没问题但没什么益处)。当希望映射不带有源和任何注解的第三方组件类时，`@Embedded` 注解就能发挥作用，但要使用正确的 `getter/setter` 方法(如常规的 `JavaBeans`)。

`@AttributeOverrides` 会选择性地重写嵌入类的属性映射；在这个示例中，重写了全部 3 个属性并且提供了不同的列名。现在可以在 `USERS` 表中存储两个 `Address` 实例了，每个实例位于一组不同的列中(可再次查看图 5-2 中的架构)。

用于组件属性的每个 `@AttributeOverride` 都“完成”了：重写的属性上的任何 `JPA` 或 `Hibernate` 注解都会被忽略。这意味着 `Address` 类上的 `@Column` 注解会被忽略——所有的 `BILLING_*` 列都可为 `NULL!` (不过，`Bean` 验证仍旧会识别组件属性上的 `@NotNull` 注解；`Hibernate` 只会重写持久化注解)。

可以进一步提升域模型的可重用性，并且通过嵌套的嵌入组件来让它的粒度更细。

5.2.4 映射嵌套的可嵌入组件

考虑一下 `Address` 类以及它如何封装地址详情：不使用简单 `city` 字符串，而是将这一详情移动到一个新的 `City` 可嵌入类中。看一下图 5-3 中修改后的域模型图。针对该映射的这一 `SQL` 架构仍然只有一个 `USERS` 表，如图 5-4 所示。

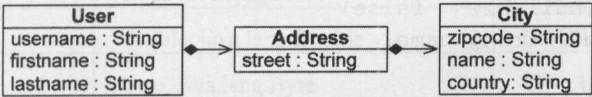


图 5-3 Address 和 City 的嵌套组合

<< Table >>	
USERS	
ID <<PK>>	
USERNAME	
FIRSTNAME	
LASTNAME	
STREET	组件列
ZIPCODE	
CITY	
COUNTRY	

图 5-4 可嵌入列会保存 Address 和 City 的详情

可嵌入类可以使用一个可嵌入属性。Address 有一个 city 属性：

路径: /model/src/main/java/org/jpwh/model/advanced/Address.java

```
@Embeddable
public class Address {

    @NotNull
    @Column(nullable = false)
    protected String street;

    @NotNull
    @AttributeOverrides(
        @AttributeOverride(
            name = "name",
            column = @Column(name = "CITY", nullable = false)
        )
    )
    protected City city;
    // ...
}
```

可嵌入的 City 类只有基本属性：

路径: /model/src/main/java/org/jpwh/model/advanced/City.java

```
@Embeddable
public class City {

    @NotNull
    @Column(nullable = false, length = 5) ← 重写 VARCHAR(255)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String name;

    @NotNull
    @Column(nullable = false)
    protected String country;
    // ...
}
```

可以继续此类嵌套，比如通过创建 Country 类来进行。所有的嵌入属性，无论它们在

组合中的位置有多深，都会被映射到所属实体表的列——这里就是 USERS 表。

可以在任何级别声明@AttributeOverrides，正如为 City 类的 name 属性所做的那样，将它映射到 CITY 列。这可以使用(所示的)Address 中的@AttributeOverride 或者根实体类 User 中的重写来实现。嵌套属性可以使用圆点符号来引用：例如，在 User#address 上，@AttributeOverride(name = "city.name")引用了 Address 的#City#name 属性。

将在稍后的第 7.2 节中回过头来介绍可嵌入组件。甚至可以映射组件的集合或者使用从组件到实体的引用。

在本章开头，讨论了基本属性以及 Hibernate 如何映射像 java.lang.String 这样的 JDK 类型，比如映射到合适的 SQL 类型。我们将继续探讨与这个类型系统有关的更多内容以及如何在较低的级别转换值。

5.3 使用转换器映射 Java 和 SQL 类型

到目前为止，已经假定，在映射 java.lang.String 属性时 Hibernate 会选择正确的 SQL 类型。然而，什么是 Java 和 SQL 类型之间的正确映射，并且如何控制它呢？

5.3.1 内置类型

所有 JPA 提供程序都必须支持 Java 到 SQL 类型转换的最小集合；在 5.1 节中看到过这个代码清单。Hibernate 支持所有这些映射，以及非标准但实际很有用的一些附加适配器。首先，介绍 Java 基元及其 SQL 等效项。

1. 基元和数值类型

表 5-1 中所示的内置类型可以将 Java 基元及其包装器映射到合适的 SQL 标准类型。其中还包含了其他一些数值类型。

表 5-1 映射到 SQL 标准类型的 Java 基元类型

名 称	Java 类型	ANSISQL 类型
integer	int, java.lang.Integer	INTEGER
long	long, java.lang.Long	BIGINT
short	short, java.lang.Short	SMALLINT
float	float, java.lang.Float	FLOAT
double	double, java.lang.Double	DOUBLE
byte	byte, java.lang.Byte	TINYINT
boolean	boolean, java.lang.Boolean	BOOLEAN
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC

这些名称都是 Hibernate 专用的；稍后在自定义类型映射时会需要用到它们。

你可能注意到了，你的 DBMS 产品不支持提到的一些 SQL 类型。这些 SQL 类型名称是 ANSI 标准类型名称。大多数 DBMS 供应商都会忽略此部分 SQL 标准，通常是因为他们的遗留类型系统领先于该标准。但 JDBC 提供了供应商专有数据类型的部分抽象，以允许 Hibernate 在执行 INSERT 和 UPDATE 等 DML 语句时使用 ANSI 标准类型。对于产品特有的架构生成，Hibernate 会使用所配置的 SQL 方言将 ANSI 标准类型转译成合适的供应商特定类型。这意味着，如果让 Hibernate 创建架构，通常不必担心 SQL 数据类型。

如果有一个现有的架构和/或需要知道 DBMS 的原生数据类型，则可以查看所配置的 SQL 方言的源。例如，Hibernate 附带的 H2Dialect 包含了从 ANSI NUMERIC 类型到供应商专有的 DECIMAL 类型的映射：`registerColumnType(Types.NUMERIC,"decimal ($p,$s)")`。

SQL 类型 NUMERIC 十进制支持精度和小数位。例如，对于 `BigDecimal` 属性，默认的十进制精度和小数位设置是 `NUMERIC(19,2)`。为了架构生成要重写它的话，可以在属性上应用 `@Column` 注解并设置其精度和小数位。

下面是映射到数据库中字符串的类型。

2. 字符类型

表 5-2 显示了映射字符和字符串值表示形式的类型。

表 5-2 字符和字符串值的适配器

名 称	Java 类型	ANSISQL 类型
String	java.lang.String	VARCHAR
character	char[], Character[], java.lang.String	CHAR
yes_no	boolean, java.lang.Boolean	CHAR(1), 'Y' or 'N'
true_false	boolean, java.lang.Boolean	CHAR(1), 'T' or 'F'
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Hibernate 类型系统会依据字符串值的声明长度来选取 SQL 数据类型：如果 `String` 属性是用 `@Column(length = ...)` 或者 Bean 验证的 `@Length` 注解的，则 Hibernate 会为给定的字符串大小选择合适的 SQL 数据类型。这一选择还取决于所配置的 SQL 方言。例如，对于 MySQL，Hibernate 生成架构时，最大长度为 65 535 会生成一个普通的 `VARCHAR(length)` 列。而对于 16 777 215 这一最大长度，则会生成 MySQL 专用的 `MEDIUMTEXT` 数据类型，更大的长度则会使用 `LONGTEXT`。Hibernate 用于所有 `java.lang.String` 属性的默认长度是 255，因此如果没有任何进一步的映射，`String` 属性会映射到 `VARCHAR(255)` 列。可以通过扩展 SQL 方言的类来自定义这一类型选择；阅读方言文档和源代码可以找到更多与 DBMS 产品有关的详细内容。

数据库通常会为整个数据库或者至少是全部表使用合理的(UTF-8)默认字符集来启

用文本的国际化。这是 DBMS 专有的设置。如果需要更细粒度的控制并且希望切换到 NVARCHAR、NCHAR 或 NCLOB 列类型，则可以使用@org.hibernate.annotations.Nationalized 注解属性映射。

此外，还内置了一些用于 Oracle 等具有受限类型系统的遗留数据库或 DBMS 的特殊转换器。Oracle DBMS 甚至没有真假值数据类型——关系模型所需要的唯一数据类型。因此许多现有的 Oracle 架构都使用 Y/N 或 T/F 字符来表示布尔值。或者——且这是 Hibernate 的 Oracle 方言所默认的——会预期并生成一个 NUMBER(1,0)类型的列。同样，如果希望了解从 ANSI 数据类型到供应商专有类型的所有映射，可以参考 DBMS 的 SQL 方言。

下面是映射到数据库中的日期和时间的类型。

3. 日期和时间类型

表 5-3 列出了与日期、时间和时间戳相关的类型。

表 5-3 日期和时间类型

名 称	Java 类型	ANSISQL 类型
date	java.util.Date, java.sql.Date	DATE
time	java.util.Date, java.sql.Time	TIME
timestamp	java.util.Date, java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
duration	java.time.Duration	BIGINT
instant	java.time.Instant	TIMESTAMP
localdatetime	java.time.LocalDateTime	TIMESTAMP
localdate	java.time.LocalDate	DATE
localtime	java.time.LocalTime	TIME
offsetdatetime	java.time.OffsetDateTime	TIMESTAMP
offsettime	java.time.OffsetTime	TIME
zoneddatetime	java.time.ZonedDateTime	TIMESTAMP

在域模型中，可以选择将日期和时间数据表示为 java.sql 包中定义的 java.util.Date、java.util.Calendar，或者 java.util.Date 的子类。这是个人喜好的问题，决定权在于你——要确保前后一致。你可能不希望将域模型绑定到 JDBC 包的类型。

你还可以使用 java.time 包中的 Java 8 API。注意，这是 Hibernate 特有的并且没有在 JPA2.1 中标准化。

用于 java.util.Date 属性的 Hibernate 行为一开始可能会让你惊讶：当存储 java.util.Date 时，Hibernate 不会在加载后返回 java.util.Date。它会返回 java.sql.Date、java.sql.Time 或 java.sql.Timestamp，这取决于使用 TemporalType.DATE、TemporalType.TIME 或 TemporalType.TIMESTAMP 中的哪一个来映射该属性。

从数据库加载数据时，Hibernate 必须使用 JDBC 子类，因为数据库类型的精确度比 java.util.Date 要高。java.util.Date 可精确到毫秒，而 java.sql.Timestamp 包含可以在数据库中呈现的纳秒信息。Hibernate 不会截取这一信息来转换成适合 java.util.Date 的值。如果试图使用 equals()方法比较 java.util.Date 值，那么这一 Hibernate 行为可能会导致问题；它与 java.sql.Timestamp 子类的 equals()方法并不对称。

解决方案很简单，甚至并不是 Hibernate 特有的：不要调用 aDate.equals(bDate)。应该总是通过 Unix 时间毫秒来比较日期和时间(假定不关心纳秒数据)：例如，aDate 晚于 bDate 时，则 aDate.getTime() > bDate.getTime()为 true。要当心：HashSet 这样的集合也会调用 equals()方法。不要在这样的集合中混淆 java.util.Date 和 java.sql.Date|Time|Timestamp 值。Calendar 属性就没有此类问题。如果存储 Calendar 值，Hibernate 就总是会返回使用 Calendar.getInstance()创建的 Calendar 值(实际类型取决于区域设置和时区)。

或者，可以编写自己的转换器(本章稍后将会介绍)，并且将 Hibernate 给出的所有 java.sql 时序类型实例转换成普通的 java.util.Date 实例。例如，如果从数据库加载值之后 Calendar 实例应该使用非默认时区，那么自定义转换器也就是一个好的着手点。

下面是映射到数据库中二进制数据和大型值的类型。

4. 二进制和大型值类型

表 5-4 列出了用于处理二进制数据和大型值的类型。注意只支持二进制用作标识符属性的类型。

首先，思考一下 Hibernate 如何将可能的大型值表示成二进制或文本。

表 5-4 二进制和大型值类型

名 称	Java 类型	ANSISQL 类型
Binary	byte[], java.lang.Byte[]	VARBINARY
Text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	java.io.Serializable	VARBINARY

如果持久化 Java 类中的属性是 byte[]类型，那么 Hibernate 会将它映射到 VARBINARY 列。真正的 SQL 数据类型取决于方言；例如，在 PostgreSQL 中，其数据类型是 BYTEA，而在 Oracle DBMS 中，它是 RAW。在有些方言中，使用@Column 设置的 length 也会对所选择的原生类型产生影响：例如，Oracle 中长度为 2000 及更大的 LONG RAW。

java.lang.String 属性会被映射到 SQL VARCHAR 列，char[]和 Character[]也是一样。正如前面探讨过的，有些方言会依据所声明的长度来注册不同的原生类型。

在这两种情况中，在加载保存属性变量的实体实例时，Hibernate 都会立即初始化该属性值。这在必须处理可能的大型值时并不方便，因此通常会希望重写这一默认映射。为此，JPA 规范有一个便利的快捷注解@Lob：

```
@Entity
public class Item {

    @Lob
    protected byte[] image;

    @Lob
    protected String description;
    // ...
}
```

这会将 `byte[]` 映射到 SQL 的 BLOB 数据类型并将 `String` 映射到 CLOB。遗憾的是，使用这一设计你仍旧无法得到延迟加载。Hibernate 将不得不拦截字段访问并且会在调用 `someItem.getImage()` 时加载 `image` 的字节。这一方式在编译后需要类的字节码指令，用于注入额外的代码。12.1.3 小节将探讨通过字节码指令和拦截进行延迟加载。

或者，可以在 Java 类中切换属性类型。JDBC 可直接支持定位符对象(LOB)。如果 Java 属性是 `java.sql.Clob` 或 `java.sql.Blob`，那么不使用字节码指令即可延迟加载。

```
@Entity
public class Item {

    @Lob
    protected java.sql.Blob imageBlob;

    @Lob
    protected java.sql.Clob description;
    // ...
}
```

BLOB/CLOB 意味着什么？

提出 LOB 概念的 Jim Starkey 说过，市场部门创建了术语 BLOB 和 CLOB，但它们没有任何意义。可以随意诠释它们。我们选用定位符对象来诠释它们，作为它们是帮助我们定位和访问真实事物的占位符的提示。

这些 JDBC 类包含了按需加载值的行为。当所属的实体实例被加载时，属性值就是一个占位符，且真正的值不会立即体现出来。一旦在相同的事务中访问该属性，其值就会体现出来或直接流出(到客户端)，而不会消耗临时内存：

路径：/examples/src/test/java/org/jpwh/test/advanced/LazyProperties.java

```
Item item = em.find(Item.class, ITEM_ID); // 可以直接流出该字节.....

InputStream imageDataStream = item.getImageBlob().getBinaryStream();

ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
StreamUtils.copy(imageDataStream, outputStream);
byte[] imageBytes = outputStream.toByteArray(); // .....或者将它们体现在内存中
```

缺点是域模型随后会被绑定到 JDBC；在单元测试中，无法不使用数据库连接来访问 LOB 属性。

Hibernate 特性

要创建并设置一个 Blob 或 Clob 值, Hibernate 要提供一些便利方法。这个示例会从 `InputStream` 中将 `byteLength` 字节读取到数据库中, 无须消耗临时内存:

```
Session session = em.unwrap(Session.class);
Blob blob = session.getLobHelper()
    .createBlob(imageInputStream, byteLength);
someItem.setImageBlob(blob);
em.persist(someItem);
```

← 需要原生 Hibernate API

← 需要知道希望从流中读取的字节数

最后, Hibernate 会为所有的 `java.io.Serializable` 属性类型提供回退序列。这个映射会将属性值转换成存储在 `VARBINARY` 列中的字节流。在所属实体实例被存储和加载时会进行序列化和反序列化。自然, 使用这一策略要非常小心, 因为数据的生命周期比应用程序要长。总有一天, 没人会知道数据库中的那些字节的含义是什么。对于临时数据来说, 序列化有时候很有用, 如用户偏好、登录会话数据等。

Hibernate 会依据属性的 Java 类型来选取正确的类型适配器。如果不想默认映射, 请继续阅读如何重写它。

5. 选择一种类型适配器

前面已经介绍许多适配器及其 Hibernate 名称。当要重写 Hibernate 的默认类型选择并且显式选择一种特定适配器时, 可以使用该名称:

```
@Entity
public class Item {
    @org.hibernate.annotations.Type(type = "yes_no")
    protected boolean verified = false;
}
```

相较于 `BIT`, `boolean` 现在会映射到具有 Y 或 N 值的 `CHAR` 列。

还可以使用一个自定义用户类型在 Hibernate 启动配置中全局重写一个适配器, 将在本章后续内容中了解到如何进行编写:

```
metaBuilder.applyBasicType(new MyUserType(), new String[]{"date"});
```

这个设置会重写内置的日期类型适配器并将用于 `java.util.Date` 属性的值转换委托给你的自定义实现。

将这一可扩展类型系统视作 Hibernate 的一个核心功能以及让 Hibernate 如此灵活的一个重要方面。接下来, 要更为详尽地探究该类型系统以及 JPA 自定义转换器。

5.3.2 创建自定义 JPA 转换器

该在线拍卖系统的一个新需求是多货币。展开此类变更会很复杂。必须修改数据库架构, 可能不得不将现有数据从旧架构迁移到新架构, 并且必须更新访问数据库的所有应用程序。本小节将介绍 JPA 转换器和可扩展的 Hibernate 类型系统能如何在这个过程中帮助

你，以便在应用程序和数据库之间提供一种额外、灵活的缓存区。

为了支持多货币，要在 CaveatEmptor 域模型中引入一个新的类：MonetaryAmount，如代码清单 5.7 所示。

代码清单5.7 不可变的MonetaryAmount值类型类

路径：/model/src/main/java/org/jpwh/model/advanced/MonetaryAmount.java

```
public class MonetaryAmount implements Serializable {
    protected final BigDecimal value;
    protected final Currency currency;

    public MonetaryAmount(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }

    public BigDecimal getValue() {
        return value;
    }

    public Currency getCurrency() {
        return currency;
    }

    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof MonetaryAmount)) return false;
        final MonetaryAmount monetaryAmount = (MonetaryAmount) o;
        if (!value.equals(monetaryAmount.value)) return false;
        if (!currency.equals(monetaryAmount.currency)) return false;
        return true;
    }

    public int hashCode() {
        int result;
        result = value.hashCode();
        result = 29 * result + currency.hashCode();
        return result;
    }

    public String toString() {
        return getValue() + " " + getCurrency();
    }

    public static MonetaryAmount fromString(String s) {
        String[] split = s.split(" ");
        return new MonetaryAmount(
            new BigDecimal(split[0]),
            Currency.getInstance(split[1])
        );
    }
}
```

① 值类型类是 java.io.Serializable

② 不需要专门的构造函数

③ 实现 equals() 和 hashCode()

④ 从 String 创建实例

- ❶ 这个值类型类应该是 `java.io.Serializable`：当 Hibernate 在共享的二级缓存中存储实体实例数据时(参阅 20.2 节)，它会拆卸该实体的状态。如果实体具有 `MonetaryAmount` 属性，则该属性值的序列化表示会被存储在二级缓存区域中。在从该缓存区域中检索实体数据时，其属性值会被反序列化和重新装配。
- ❷ 这个类不需要专门的构造函数。可以让其不可变，甚至使用 `final` 字段，因为你的代码会是创建实例的唯一位置。
- ❸ 应该实现 `equals()` 和 `hashCode()` 方法并“通过值”来比较货币数量。
- ❹ 货币数量需要一个 `String` 表示形式。实现 `toString()` 方法和静态方法，以便从 `String` 中创建一个实例。

接下来，要更新该拍卖域模型的其他部分并且将 `MonetaryAmount` 用于所有涉及货币的属性，如 `Item#buyNowPrice` 和 `Bid#amount`。

1. 转换基本属性值

通常就是这样，数据库方面的同事不能立即实现多货币且需要更多的时间。他们所能快速提供的就是数据库模式中一列数据类型的变更。他们会建议你在 `ITEM` 表的 `VARCHAR` 列中存储 `BUYNOWPRICE`，并且要将货币数量的货币代码附加到其字符串值。例如，要存储 `11.23 USD` 或 `99 EUR` 这样的值。

在存储数据时，必须将 `MonetaryAmount` 的实例转换成这样的 `String` 表示形式。在加载数据时，要将该 `String` 转换回到 `MonetaryAmount`。

最简单的解决方案是 `javax.persistence.AttributeConverter`，如代码清单 5.8 所示，它是 JPA 中的一个标准扩展点。

代码清单5.8 在字符串和MonetaryValue之间转换

路径：`/model/src/main/java/org/jpwh/converter/MonetaryAmountConverter.java`

```
@Converter(autoApply = true)
public class MonetaryAmountConverter
    implements AttributeConverter<MonetaryAmount, String> {

    @Override
    public String convertToDatabaseColumn(MonetaryAmount monetaryAmount)
    {
        return monetaryAmount.toString();
    }

    @Override
    public MonetaryAmount convertToEntityAttribute(String s) {
        return MonetaryAmount.fromString(s);
    }
}
```

← 默认用于 MonetaryAmount 属性

转换器必须实现 `AttributeConverter` 接口；其两个类型参数是 Java 属性的类型和数据库架构中的类型。Java 类型是 `MonetaryAmount`，而数据库类型是 `String`，通常它会映射到 SQL `VARCHAR`。必须使用 `@Converter` 注解这个类或者像 `orm.xml` 元数据中那样声明它。启用

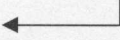
autoApply 之后，域模型中的所有 MonetaryAmount 属性，无论是实体还是可嵌入类，现在无须进一步映射就能由这个转换器自动处理(不要被 AttributeConverter 接口的 convertToEntity-Attribute()方法所干扰；它并非最佳的名称)。

域模型中的这个 MonetaryAmount 属性的例子是 Item#buyNowPrice:

路径: /model/src/main/java/org/jpwh/model/advanced/converter/Item.java

```
@Entity
public class Item {
    @NotNull
    @Convert(
        converter = MonetaryAmountConverter.class,
        disableConversion = false)
    @Column(name = "PRICE", length = 63)
    protected MonetaryAmount buyNowPrice;
    // ...
}
```

可选: 启用了 autoApply



@Convert 注解是可选的：应用它重写或禁用特定属性的转换器。@Column 将所映射的数据库列重命名为 PRICE；默认的名称是 BUYNOWPRICE。为了自动的架构生成，可以将它定义为具有 63 个字符长度的 VARCHAR。

稍后，当 DBA 升级了数据库架构并且提供了用于货币数量值和币种的独立列时，只需要修改应用程序中的几处即可。从项目中移除 MonetaryAmountConverter 并且用 @Embeddable 注解 MonetaryAmount；然后它会自动映射到两个数据库列。如果架构中的一些表还没有升级，有选择地启用和禁用转换器也很容易。

刚刚编写的转换器是用于 MonetaryAmount 的，它是域模型中一个新的类。转换器并不局限于自定义类：甚至可以重写 Hibernate 的内置类型适配器。例如，可以为域模型中的一些甚或所有 java.util.Date 属性创建自定义转换器。

可以将转换器应用到实体类的属性，就像上一个示例中的 Item#buyNowPrice 一样。还可以将它们应用到可嵌入类的属性。

2. 转换组件的属性

在本章中已经提供了细粒度域模型的例子。早些时候，隔离了 User 的地址信息并且映射了 Address 可嵌入类。下面继续这一过程并且用一个抽象的 Zipcode 类来介绍继承，如图 5-5 所示。

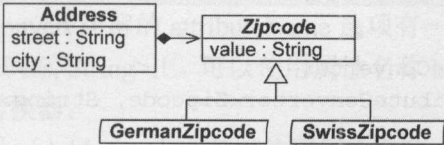


图 5-5 抽象的 Zipcode 类有两个具体的子类

该 Zipcode 类不太重要，但不要忘记实现值的相等性：

路径: /model/src/main/java/org/jpwh/model/advanced/converter/Zipcode.java

```
abstract public class Zipcode {

    protected String value;

    public Zipcode(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Zipcode zipcode = (Zipcode) o;
        return value.equals(zipcode.value);
    }

    @Override
    public int hashCode() {
        return value.hashCode();
    }

}
```

现在可以封装域子类，德国和瑞士邮政编码之间的区别以及所有的处理：

路径: /model/src/main/java/org/jpwh/model/advanced/converter/GermanZipcode.java

```
public class GermanZipcode extends Zipcode {

    public GermanZipcode(String value) {
        super(value);
    }

}
```

还没有在子类中实现任何特殊处理。从最明显的区别开始：德国邮编是五位数字长度，而瑞士是四位。自定义转换器会负责这一处理：

路径: /model/src/main/java/org/jpwh/converter/ZipcodeConverter.java

```
@Converter
public class ZipcodeConverter
    implements AttributeConverter<Zipcode, String> {

    @Override
    public String convertToDatabaseColumn(Zipcode attribute) {
        return attribute.getValue();
    }

    @Override
```

```

public Zipcode convertToEntityAttribute(String s) {
    if (s.length() == 5)
        return new GermanZipcode(s);
    else if (s.length() == 4)
        return new SwissZipcode(s);
    throw new IllegalArgumentException(
        "Unsupported zipcode in database: " + s
    );
}
}

```

如果进行到此处，则要考虑清理数据库……或创建 InvalidZipCode 子类并在此处返回它

Hibernate 会在存储属性值时调用该转换器的 `convertToDatabaseColumn()` 方法；你要返回一个 `String` 表示形式。架构中的列是 `VARCHAR`。在加载一个值的时候，要检验其长度并且创建 `GermanZipcode` 或 `SwissZipcode` 实例。这是一个自定义类型区别例程；可以选取给定值的 `Java` 类型。

现在将此转换器应用到一些 `Zipcode` 属性上——例如，`User` 的可嵌入 `homeAddress`：

路径：/model/src/main/java/org/jpwh/model/advanced/converter/User.java

```

@Entity
@Table(name = "USERS")
public class User implements Serializable {

    @Convert(
        converter = ZipcodeConverter.class,
        attributeName = "zipcode"
    )
    protected Address homeAddress;
    // ...
}

```

使用 @Converts 分组多个属性转换

或者用于嵌套可嵌入的 city.zipcode

`attributeName` 声明了可嵌入 `Address` 类的 `zipcode` 属性。这一设置支持将圆点语法用于属性路径；如果 `zipcode` 不是 `Address` 类的属性而是一个嵌套可嵌入 `City` 类的属性(正如本章前面内容所示)，则要使用其嵌套路径 `city.zipcode` 来引用它。

如果单个可嵌入属性上需要几个 `@Convert` 注解，例如，要转换 `Address` 的几个属性，那么可以将它们分组到一个 `@Converts` 注解中。还可以将转换器应用到集合与映射的值，如果它们的值和/或键是基本类型或可嵌入类型。例如，可以在持久化 `Set<Zipcode>` 上添加 `@Convert` 注解。稍后在第7章中，我们将介绍如何使用 `@ElementCollection` 映射持久化集合。

对于持久化映射，`@Convert` 注解的 `attributeName` 选项有一些特殊语法：

- 在持久化 `Map<Address, String>` 上，可以使用属性名称 `key.zipcode` 为每个映射键的 `zipcode` 属性应用转换器。
- 在持久化 `Map<String, Address>` 上，可以使用属性名称 `value.zipcode` 为每个映射值的 `zipcode` 属性应用转换器。
- 在持久化 `Map<Zipcode, String>` 上，可以使用属性名称 `key` 为每个映射条目的键应用转换器。

- 在持久化 Map<String, Zipcode>上，不设置任何 attributeName，就可以为每个映射条目的值应用转换器。

像之前一样，如果可嵌入类是嵌套的，则属性名称可以是圆点分隔的路径；可以编写 key.city.zipcode 来引用 City 类的 zipcode 属性，与 Address 类组合起来使用。

JPA 转换器的一些约束如下：

- 不能将它们应用到实体的标识符或版本属性。
- 不应该在使用 @Enumerated 或 @Temporal 映射的属性上应用转换器，因为这些注解已经声明了必须发生何种转换。如果希望为枚举或日期/时间属性应用自定义转换器，则不要使用 @Enumerated 或 @Temporal 注解它们。
- 可以在 hbm.xml 文件中将转换器应用到属性映射，但必须为其名称添加前缀：type="converter:qualified.ConverterName"。

回到 CaveatEmptor 中的多货币支持。数据库管理员再次修改了架构并且要求更新应用程序。

Hibernate 特性

5.3.3 使用 UserTypes 扩展 Hibernate

终于，已经将新的列添加到了数据库架构，以支持多货币。现在 ITEM 表有了一个 BUYNOWPRICE_AMOUNT 以及用于货币数量的独立列 BUYNOWPRICE_CURRENCY。还有 INITIALPRICE_AMOUNT 和 INITIALPRICE_CURRENCY 列。必须将这些列映射到 Item 类、buyNowPrice 和 initialPrice 的 MonetaryAmount 属性。

理想情况下，不会希望变更域模型；属性已经使用了 MonetaryAmount 类。遗憾的是，标准 JPA 转换器不支持值从/到多个列的转换。JPA 转换器的另一个约束是与查询引擎的集成。不能编写以下查询：select i from Item i where i.buyNowPrice.amount > 100。感谢 5.3.2 小节的转换器，Hibernate 知道如何将 MonetaryAmount 转换为字符串，反之亦然。它并不知道 MonetaryAmount 有 amount 属性，因此它无法解析这样的查询。

一个简单的解决方案是将 MonetaryAmount 映射为 @Embeddable，如本章早前介绍的 Address 类所那样。MonetaryAmount 的每个属性——amount 和 currency——都会映射到其各自的数据库列。

不过，数据库管理员给需求增加了变数：因为其他旧有应用程序也会访问该数据库，所以必须在将货币数量存储到数据库之前将每个数量转换成目标货币。例如，Item#buyNowPrice 应该以美元存储，而 Item#initialPrice 应该以欧元存储(虽然该例看起来有点牵强，但在现实环境中肯定会看到更糟糕的情况。共享数据库架构的演化成本会很高，但又是必须的，因为数据库的生命周期比应用程序要久)。Hibernate 提供了一个原生转换器 API：允许更多详细细节和低级别自定义访问的扩展点。

1. 扩展点

可以在 org.hibernate.usertype 包中找到用于其类型系统的 Hibernate 扩展接口。有以下

接口可用:

- **UserType**——可以通过与普通 JDBC 的 `PreparedStatement`(在存储数据时)和 `ResultSet`(在加载数据时)交互来转换值。通过实现此接口,还可以控制 Hibernate 如何对值进行缓存以及脏检查。`MonetaryAmount` 的适配器必须实现此接口。
- **CompositeUserType**——它扩展了 `UserType`,为 Hibernate 提供了更多与适配过的类有关的详细信息。可以告知 Hibernate `MonetaryAmount` 组件有两个属性: `amount` 和 `currency`。然后可以在查询中使用圆点符号引用这些属性:例如, `select avg(i.buyNowPrice.amount) from Item i`。
- **ParameterizedUserType**——它提供了对映射中适配器的设置。必须为 `MonetaryAmount` 转换实现此接口,因为在有些映射中你希望将货币数量转换成美元而在其他映射中转换成欧元。在映射一个属性时,只需要编写单个适配器并且可以定制其行为。
- **DynamicParameterizedType**——这个更强大的设置 API 允许访问适配器中的动态信息,如映射列和表名称。也可以使用它来替代 `ParameterizedUserType`;没有额外的开销或复杂性。
- **EnhancedUserType**——这是用于标识符属性和鉴别器的可选接口。与 JPA 转换器不同, Hibernate 中的 `UserType` 可以是用于任何类型实体属性的适配器。因为 `MonetaryAmount` 不会是标识符属性或鉴别器类型,你不需要它。
- **UserVersionType**——这是用于版本属性适配器的可选接口。
- **UserCollectionType**——这个很少用到的接口用于实现自定义集合。必须实现它来持久化非 JDK 集合并保留额外的语义。

用于 `MonetaryAmount` 的自定义类型适配器将实现其中几个接口。

2. 实现 UserType

由于 `MonetaryAmountUserType` 是一个大类,所以分几个步骤来检验它。下面是要实现的接口。

路径: `/model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java`

```
public class MonetaryAmountUserType
    implements CompositeUserType, DynamicParameterizedType {
    // ...
}
```

首先要实现 `DynamicParameterizedType`。需要通过检验映射参数来为该转换配置目标货币:

路径: `/model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java`

```
protected Currency convertTo;

public void setParameterValues(Properties parameters) {

    ParameterType parameterType =
        (ParameterType) parameters.get(PARAMETER_TYPE);
    String[] columns = parameterType.getColumns();
}
```

访问动态参数

```
String table = parameterType.getTable();
Annotation[] annotations = parameterType.getAnnotationsMethod();

String convertToParameter = parameters.getProperty("convertTo");
this.convertTo = Currency.getInstance(
    convertToParameter != null ? convertToParameter : "USD"
);
}
```

- 可以在此处访问一些动态参数，如所映射的列的名称、所映射的(实体)表、甚或所映射属性的字段/getter 方法上的注解。不过，这个示例中不需要它们。
- 在将一个值保存到数据库中时，只要使用 `convertTo` 参数判定目标货币即可。如果该参数还未设置，则默认为美元。

接下来，是所有 `UserType` 都必须实现的一些脚手架代码：

路径：/model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```
public Class returnedClass() {
    return MonetaryAmount.class;
}

public boolean isMutable() {
    return false;
}

public Object deepCopy(Object value) {
    return value;
}

public Serializable disassemble(Object value,
    SessionImplementor session) {
    return value.toString();
}

public Object assemble(Serializable cached,
    SessionImplementor session, Object owner) {
    return MonetaryAmount.fromString((String) cached);
}

public Object replace(Object original, Object target,
    SessionImplementor session, Object owner) {
    return original;
}

public boolean equals(Object x, Object y) {
    return x == y || !(x == null || y == null) && x.equals(y);
}

public int hashCode(Object x) {
    return x.hashCode();
}
}
```

①适配类

②启用优化

③复制值

④返回可序列化的表示形式

⑤创建 MonetaryAmount 实例

⑥返回 original 的副本

⑦判定是否修改过值

- ❶ returnedClass 方法会适配指定的类，在这个例子中是 MonetaryAmount。
- ❷ 如果 Hibernate 知道 MonetaryAmount 是不可变的，则它可以启用一些优化。
- ❸ 如果 Hibernate 必须得到值的副本，则它会调用此方法。对于 MonetaryAmount 等简单不可变类来说，可以返回指定实例。
- ❹ Hibernate 会在全局共享二级缓存中存储值时调用 disassemble。需要返回一个 Serializable 表示形式。对于 MonetaryAmount，String 表示形式是一个简单的解决方案。或者，由于 MonetaryAmount 是 Serializable，所以可以直接返回它。
- ❺ Hibernate 会在从全局共享二级缓存中读取该序列化表示形式时调用此方法。要从 String 表示形式创建 MonetaryAmount 实例。或者，如果存储一个序列化 MonetaryAmount，则可以直接返回它。
- ❻ 这是在 EntityManager#merge() 操作期间调用的。需要返回原始对象的副本。或者，如果值类型是不可变的，如 MonetaryAmount，则可以返回该原始对象。
- ❼ Hibernate 会使用值相等性来判定值是否被修改以及数据库是否需要更新。你要依赖已经在 MonetaryAmount 类上编写的相等性例程。

适配器的实际运行发生在值被加载和存储时，就像以下方法所实现的：

路径：/model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```
public Object nullSafeGet(ResultSet resultSet, ❶ 读取 ResultSet
                        String[] names,
                        SessionImplementor session,
                        Object owner) throws SQLException {

    BigDecimal amount = resultSet.getBigDecimal(names[0]);
    if (resultSet.wasNull())
        return null;
    Currency currency =
        Currency.getInstance(resultSet.getString(names[1]));
    return new MonetaryAmount(amount, currency);
}

public void nullSafeSet(PreparedStatement statement, ❷ 存储 MonetaryAmount
                        Object value,
                        int index,
                        SessionImplementor session) throws SQLException {

    if (value == null) {
        statement.setNull(
            index,
            StandardBasicTypes.BIG_DECIMAL.sqlType());
        statement.setNull(
            index + 1,
            StandardBasicTypes.CURRENCY.sqlType());
    } else {
        MonetaryAmount amount = (MonetaryAmount) value;
        MonetaryAmount dbAmount = convert(amount, convertTo);
        statement.setBigDecimal(index, dbAmount.getValue());
        statement.setString(index + 1, convertTo.getCurrencyCode());
    }
}
```

保存时，转换到
目标货币

```

    }
}
protected MonetaryAmount convert(MonetaryAmount amount,
                                   Currency toCurrency) {
    return new MonetaryAmount(
        amount.getValue().multiply(new BigDecimal(2)),
        toCurrency
    );
}

```

③ 转换货币

- ❶ 在必须从数据库检索 MonetaryAmount 值时，就会调用它来读取 ResultSet。要使用查询结果中给定的 amount 和 currency 值并且创建 MonetaryAmount 的新实例。
- ❷ 当 MonetaryAmount 值必须存储到数据库中时，就会调用它。要将该值转换成目标货币，然后在提供的 PreparedStatement 上设置 amount 和 currency(除非 MonetaryAmount 为 null，那样要调用 setNull()来准备语句)。
- ❸ 此处可以实现所需要的任意种类货币转换。对于这个示例，要让该值翻倍，就能轻易测试转换是否成功了。在真实的应用程序中你必须使用一个真实货币转换器替换该代码。它并非 Hibernate UserType API 的一个方法。

最后，以下是 CompositeUserType 接口所需要的方法，提供 MonetaryAmount 属性的详情，以便 Hibernate 可以将该类集成到查询引擎：

路径：/model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```

public String[] getPropertyNames() {
    return new String[]{"value", "currency"};
}

public Type[] getPropertyTypes() {
    return new Type[]{
        StandardBasicTypes.BIG_DECIMAL,
        StandardBasicTypes.CURRENCY
    };
}

public Object getPropertyValue(Object component,
                                int property) {
    MonetaryAmount monetaryAmount = (MonetaryAmount) component;
    if (property == 0)
        return monetaryAmount.getValue();
    else
        return monetaryAmount.getCurrency();
}

public void setPropertyValue(Object component,
                              int property,
                              Object value) {
    throw new UnsupportedOperationException(
        "MonetaryAmount is immutable"
    );
}

```

现在 `MonetaryAmountUserType` 就完成了，已经可以在 `@org.hibernate.annotations.Type` 中用其完全限定类名称使用它进行映射了，就像 5.3.1 小节“5.选择一种类型适配器”中所介绍的那样。这个注解还支持参数，因此可以将 `convertTo` 参数设置为目标货币。

但建议创建类型定义，使用一些参数绑定适配器。

3. 使用类型定义

需要一个转换到美元的适配器和一个转换到欧元的适配器。如果将这些参数作为一个类型定义声明一次，就不必在属性映射中重复它们了。类型定义的合适位置是包元数据，在 `package-info.java` 文件中：

路径：/model/src/main/java/org/jpwh/converter/package-info.java

```
@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name = "monetary_amount_usd",
        typeClass = MonetaryAmountUserType.class,
        parameters = {@Parameter(name = "convertTo", value = "USD")}
    ),
    @org.hibernate.annotations.TypeDef(
        name = "monetary_amount_eur",
        typeClass = MonetaryAmountUserType.class,
        parameters = {@Parameter(name = "convertTo", value = "EUR")}
    )
})
package org.jpwh.converter;

import org.hibernate.annotations.Parameter;
```

现在已经准备好在映射中使用这些适配器了，可以使用名称 `monetary_amount_usd` 和 `monetary_amount_eur`。

映射 `Item` 的 `buyNowPrice` 和 `initialPrice`：

路径：/model/src/main/java/org/jpwh/model/advanced/usertype/Item.java

```
@Entity
public class Item {

    @NotNull
    @org.hibernate.annotations.Type(
        type = "monetary_amount_usd"
    )
    @org.hibernate.annotations.Columns(columns = {
        @Column(name = "BUYNOWPRICE_AMOUNT"),
        @Column(name = "BUYNOWPRICE_CURRENCY", length = 3)
    })
    protected MonetaryAmount buyNowPrice;

    @NotNull
    @org.hibernate.annotations.Type(
        type = "monetary_amount_eur"
    )
    protected MonetaryAmount initialPrice;
```



```
@org.hibernate.annotations.Columns(columns = {
    @Column(name = "INITIALPRICE_AMOUNT"),
    @Column(name = "INITIALPRICE_CURRENCY", length = 3)
})
protected MonetaryAmount initialPrice;
// ...
}
```

如果 `UserType` 只为单个列转换值，则无须 `@Column` 注解。不过，`MonetaryAmountUserType` 会访问两个列，因此需要在属性映射中显式声明两列。由于 JPA 不支持单个属性上的多个 `@Column` 注解，所以必须使用专有的 `@org.hibernate.annotations.Columns` 注解来对它们进行分组。注意注解的顺序现在很重要！重新检查用于 `MonetaryAmountUserType` 的代码；许多操作都依赖数组的索引访问。访问 `PreparedStatement` 或 `ResultSet` 时的顺序与映射中所声明列的顺序相同。还要注意列的数量与选择 `UserType` 还是 `CompositeUserType` 无关——仅与是否希望公开用于查询的值类型属性有关。

使用 `MonetaryAmountUserType`，就已经扩展了 Java 域模型和 SQL 数据库架构之间的缓冲。这两种表示形式对于变更来说现在都很健壮，并且你甚至可以应对异乎寻常的需求，而无须修改域模型类的实质。

5.4 本章小结

- 讨论了实体类基本和嵌入属性的映射。
- 介绍了如何重写基本映射、如何变更映射列的名称，及如何使用派生、默认、顺序和枚举属性。
- 介绍了可嵌入组件类，以及如何创建细粒度域模型。
- 可以用一个组合来映射几个 Java 类的属性，例如将 `Address` 和 `City` 映射到一个实体表。
- 介绍了 Hibernate 如何选择 Java 到 SQL 类型的转换器，以及 Hibernate 中内置了哪些类型。
- 使用标准 JPA 扩展接口编写了用于 `MonetaryAmount` 类的一个自定义类型转换器，然后使用原生 Hibernate `UserType` API 编写了一个低级别适配器。

本章内容简介:

- 继承关系映射策略
- 多态关联

到目前为止,有意未过多谈论继承关系映射。将类的层次结构映射到表是一个复杂的问题,本章将介绍各种策略。

用于将类映射到数据库表的基本策略可能是“将一个表用于每一个实体持久化类”。这一方式听起来相当简单且实际上能很好地发挥作用,直到遇到继承关系。

在面向对象和关系世界之间,继承关系是如此明显的一种结构上的不匹配,因为面向对象系统的模型既是 is a(是……)关系,又是 has a(有……)关系。基于 SQL 的模型所提供的仅仅是 has a 关系;SQL 数据库管理系统不支持类型继承——甚至当其可用时,它通常也是专有或者不完整的。

有 4 种不同的策略可用于表示继承关系的层次结构:

- 每个具体类使用一个表并且使用默认的运行时多态行为。
- 每个具体类使用一个表但完全舍弃 SQL 架构的多态和继承关系。将 SQL UNION 查询用于运行时多态行为。
- 每个类层次结构使用一个表:通过反规范化 SQL 架构来启用多态并且依赖基于行的区别来判定超类型/子类型。
- 每个子类使用一个表:将 is a(继承)关系表示为 has a(外键)关系,并且使用 SQL JOIN 操作。

本章使用自上而下的方法,假定正开始处理一个域模型并且试图派生一个新的 SQL 架构。所描述的该映射策略同样适用于使用自下而上的方法处理已有数据库表的情况。在这个过程中将介绍一些技巧,以帮助处理不完美的表布局。

6.1 每个带有隐式多态的具体类使用一个表

假定坚持使用所建议的最简单方法:为每个具体类准确地使用一个表。可将类的所有属性映射到这个表的列,包括继承的属性,如图 6-1 所示。

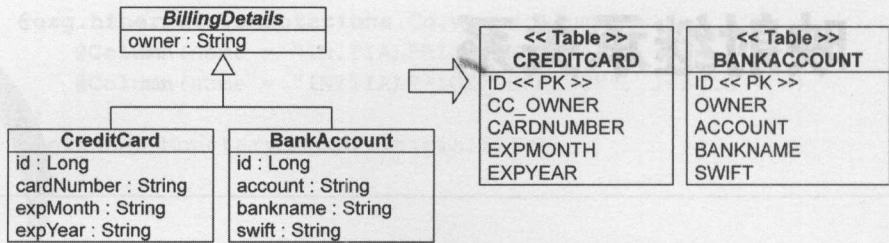


图 6-1 将所有具体类映射到一个独立的表

如果正依赖于这一隐式多态，则要像往常一样使用@Entity 映射具体类。默认情况下，超类的属性会被忽略并且不会持久化！必须使用@MappedSuperclass 注解该超类，以便在具体子类表中启用其属性的嵌入；参看以下代码清单 6.1。

代码清单6.1 映射具有隐式多态的BillingDetails(抽象超类)

路径: /model/src/main/java/org/jpwh/model/inheritance/mappedsuperclass/
BillingDetails.java

```
@MappedSuperclass
public abstract class BillingDetails {

    @NotNull
    protected String owner;
    // ...
}
```

现在映射具体的子类。参见代码清单 6.2。

代码清单6.2 映射CreditCard(具体子类)

路径: /model/src/main/java/org/jpwh/model/inheritance/mappedsuperclass/CreditCard.
java

```
@Entity
@AttributeOverride(
    name = "owner",
    column = @Column(name = "CC_OWNER", nullable = false))
public class CreditCard extends BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;
    // ...
}
```

BankAccount 类的映射看起来相同，所以这里就不介绍了。

可在一个子类中使用 `@AttributeOverride` 注解或在几个子类中使用 `@AttributeOverrides` 重写来自超类的列映射。上一个示例将 `CREDITCARD` 表中的 `OWNER` 列重命名为 `CC_OWNER`。

可以在超类中声明标识符属性，使用一个共享的列名称并且将生成器策略用于所有子类，这样就不必重复它了。在示例中没有这样做，是为了说明它是可选的。

隐式继承关系映射的主要问题在于，它不能很好地支持多态关联。在数据库中，通常要将关联表示为外键关系。在图 6-1 所示的架构中，如果子类都被映射到不同的表，则对其超类(抽象 `BillingDetails`)的多态关联就不能表示为简单外键关系。不能使用外键“引用 `BILLINGDETAILS`”来映射另一个实体——没有这样的表。这在域模型中会产生问题，因为 `BillingDetails` 是与 `User` 相关联的；`CREDITCARD` 和 `BANKACCOUNT` 表都需要对 `USERS` 表的外键引用。这些问题都无法轻易解决，所以应该考虑一种替代映射策略。

返回匹配所查询类的接口的所有类实例的多态查询也会产生问题。`Hibernate` 必须将针对超类的查询以几个 `SQL SELECT` 的形式执行，每个对应一个子类。`JPA` 查询 `select bd from BillingDetails bd` 需要两个 `SQL` 语句：

```
select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT
from
    BANKACCOUNT
select
    ID, CC_OWNER, CARDNUMBER, EXPMONTH, EXPYEAR
from
    CREDITCARD
```

`Hibernate` 会为每个具体子类使用独立的 `SQL` 查询。另一方面，针对具体类的查询都很平常并且会很好地执行——`Hibernate` 仅使用其中一个语句。

使用这一映射策略更进一步的概念问题是，不同表的几个不同列完全共享相同语意。这使架构演化更复杂。例如，重命名或修改超类属性的类型会引发对多个表中多个列的修改。`IDE` 所提供的许多标准重构操作都需要手动调整，因为自动化过程通常不能解释像 `@AttributeOverrides` 这样的内容。它还使实现应用到所有子类的数据库完整性约束更困难。

建议(仅)将此方法用于类层次结构的顶层，其中通常不需要多态，而且未来不太可能修改超类。它并不能很好地适用于 `CaveatEmptor` 域模型，其中查询和其他实体引用了 `BillingDetails`。

有了 `SQL UNION` 操作的帮助，就可以解决使用多态查询和关联的大多数问题。

6.2 每个带有联合的具体类使用一个表

首先，考虑一个将 `BillingDetails` 用作抽象类(或接口)的联合子类映射，就像 6.1 节一样。这种情况下，再次使用这两个表：`CREDITCARD` 和 `BANKACCOUNT`，并在这两个表中复制超类列。这里新内容是被称为 `TABLE_PER_CLASS` 的一个继承策略，它是在超类上声明的，如代码清单 6.3 所示。

代码清单6.3 使用TABLE_PER_CLASS映射BillingDetails

路径: /model/src/main/java/org/jpwh/model/inheritance/tableperclass/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String owner;
    // ...
}
```

数据库标识符及其映射必须在超类中提供,以便在所有子类及其表中共享它。这不再像用于前面的映射策略那样是可选的。CREDITCARD 和 BANKACCOUNT 表都有 ID 主键列。所有具体类映射继承都会来自超类(或接口)的持久化属性。每个子类上的@Entity 注解就完全满足需要了。见代码清单 6.4。

代码清单6.4 映射CreditCard

路径: /model/src/main/java/org/jpwh/model/inheritance/tableperclass/Credit-Card.java

```
@Entity
public class CreditCard extends BillingDetails {

    @NotNull
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;
    // ...
}
```

要牢记 SQL 架构仍旧不清楚该继承关系;表看起来完全一样,如图 6-1 所示。

注意, JPA 标准指定了 TABLE_PER_CLASS 是可选的,所以并非所有的 JPA 实现都会支持它。该实现也是独立于供应商的——在 Hibernate 中,它等同于老的原生 Hibernate XML 元数据中的<union-subclass>映射(如果从来没有使用过原生 Hibernate XML 文件,也不必担心这一点)。

如果 BillingDetails 是具体的,还需要一个额外的表来保存实例。必须再次强调,数据库表之间仍无关系,只是实际上它们有一些(许多)类似的列。

如果检验多态查询,那么这一映射策略的优势就会更明显。例如,查询 select bd from BillingDetailsbd 会生成以下 SQL 语句:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, CLAZZ_
from
    ( select
        ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
        null as ACCOUNT,
        null as BANKNAME,
        null as SWIFT,
        1 as CLAZZ_
    from
        CREDITCARD
    union all
    select
        id, OWNER,
        null as EXPMONTH,
        null as EXPYEAR,
        null as CARDNUMBER,
        ACCOUNT, BANKNAME, SWIFT,
        2 as CLAZZ_
    from
        BANKACCOUNT
    ) as BILLINGDETAILS
```

这个 SELECT 是用了一个 FROM 子句的子查询来从所有具体类表中检索 `BillingDetails` 的所有实例。这些表是用 UNION 操作符来联合的，并且会将一个文字(本例中是 1 和 2)插入到中间结果中；Hibernate 会读取该结果，根据特定行的数据来实例化正确的类。联合要求被合并的查询投影到相同的列；因此，必须使用 NULL 来拼凑和填充不存在的列。你可能会问，这个查询是否真的会比两个独立语句执行得更好。这里可以让数据库优化器找出最佳的执行计划来从几个表中合并行，而非像 Hibernate 的多态加载器引擎那样在内存中合并两个结果集。

另一个更为重要的优势是处理多态关联的能力；例如，从 `User` 到 `BillingDetails` 的关联映射现在就是可行的了。Hibernate 可以使用 UNION 查询来将单个表模拟为关联映射的目标。本章后续内容将详尽地介绍这一主题。

到目前为止，已经探讨过的继承关系映射策略都不需要额外考虑与 SQL 架构有关的事情。这一情况将在下一个策略发生变化。

6.3 每个类层次结构使用一个表

可将整个类层次结构映射到单个表。这个表包括用于该层次结构中所有类的所有属性的列。一个额外类型识别器列或公式的值会标识出特定行所表示的具体子类。图 6-2 显示了这一方法。

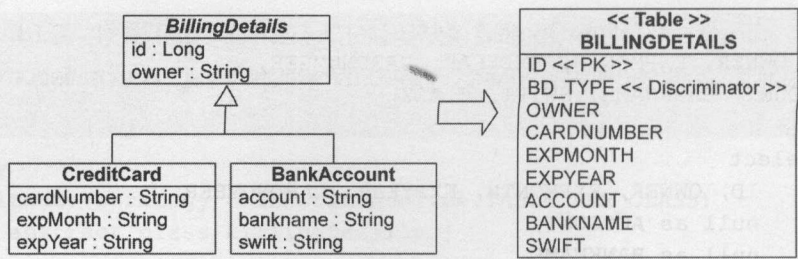


图 6-2 将整个类层次结构映射到单个表

这个映射策略在性能和简单性方面都是最优的。它是表示多态的最佳执行方式——多态和非多态查询都能很好地执行——甚至手动编写查询也很容易。不使用复杂联结或联合也能得到专门的报告。架构演化简单明了。

这里有一个主要的问题：数据完整性。必须将用于由子类声明的属性的列声明为可为空。如果每个子类定义几个非空属性，那么从数据正确性的角度看，缺少 NOT NULL 约束可能就会是严重的问题。想象一下，需要信用卡的一个过期日期，但数据库架构不能强制实现这一规则，因为表的所有列都可为 NULL。简单的应用程序编程错误就会导致产生无效的数据。

另一个重要的问题是规范化。已经在非键列之间创建了函数依赖，违反了第三范式。和往常一样，出于性能原因的反规范化会是一种误导，因为它为了眼前的获益而牺牲了长期的稳定性、可维护性及数据完整性，而这还可以通过 SQL 执行计划的正确优化来达成(换句话说，可以寻求 DBA 的帮助)。

使用 SINGLE_TABLE 继承策略来创建每个类层次结构一个表的映射，如代码清单 6.5 所示。

代码清单6.5 用SINGLE_TABLE映射BillingDetails

路径: /model/src/main/java/org/jpwh/model/inheritance/singletable/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "BD_TYPE")
public abstract class BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    @Column(nullable = false)
    protected String owner;
    // ...
}
```

← 为了架构生成而被 Hibernate 忽略！

该继承关系层次结构的根类 `BillingDetails` 被自动映射到表 `BILLINGDETAILS`。超类的共享属性在架构中可以为 NOT NULL；每个子类实例都必须有值。Hibernate 实现的一个奇怪之处在于要求使用 `@Column` 声明可空性，因为 Hibernate 会在生成数据库架构时忽略 Bean 验证的 `@NotNull`。

必须添加一个特殊的识别器列来区分每一行表示的是什么。这并非实体的属性；它是 Hibernate 内部使用的。其列名称是 `BD_TYPE`，而值是字符串——在这个例子中是“CC”或“BA”。Hibernate 会自动设置和检索该识别器值。

如果不在超类中指定识别器列，其名称会默认为 `DTYPE` 且值为字符串。该继承关系层次结构中所有的具体类都可以使用一个识别器值，如 `CreditCard`。见代码清单 6.6。

代码清单6.6 映射CreditCard

路径: /model/src/main/java/org/jpwh/model/inheritance/singletable/CreditCard.java

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
```

```
    @NotNull
```

```
    protected String cardNumber;
```

← 为了 DDL 生成而被 Hibernate 所忽略!

```
    @NotNull
```

```
    protected String expMonth;
```

```
    @NotNull
```

```
    protected String expYear;
```

```
    // ...
}
```

Hibernate 特性

没有明确的识别器值，如果使用 Hibernate XML 文件，则 Hibernate 默认会使用完全限定的类名称，并且如果使用注解或 JPA XML 文件，则使用简单实体名称。注意，JPA 不会为非字符串识别器类型指定默认名称；每个持久化提供程序都可以使用不同的默认名称。因此，应该总是为具体类指定识别器值。

使用 `@Entity` 注解每个子类，然后将子类的属性映射到 `BILLINGDETAILS` 表中的列。记住，架构中不允许 `NOT NULL` 约束，因为 `BankAccount` 实例不会使用 `expMonth` 属性，而 `EXPMONTH` 列必须为该行使用 `NULL`。Hibernate 会为架构 DDL 生成而忽略 `@NotNull`，但在插入行之前，它会在运行时监测它。这有助于避免编程错误；你不会希望偶然保存不带有过期日期的信用卡数据(另外，欠缺正确行为的应用程序当然仍旧会在数据库中存储错误的数据)。

Hibernate 会为 `select bd from BillingDetailsbd` 生成以下 SQL:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, BD_TYPE
from
    BILLINGDETAILS
```

要查询 `CreditCard` 子类，Hibernate 会在识别器列上添加一个限制:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
```

```
from
    BILLINGDETAILS
where
    BD_TYPE='CC'
```

有时候，尤其是在遗留架构中，无法在实体表中自由地包含一个额外的识别器列。这种情况下，可应用一个表达式为每一行计算一个识别器值。用于识别的公式并非 JPA 规范的一部分，但 Hibernate 有一个扩展注解 `@DiscriminatorFormula`。见代码清单 6.7。

代码清单 6.7 使用 `@DiscriminatorFormula` 映射 `BillingDetails`

路径: `/model/src/main/java/org/jpwh/model/inheritance/singletableformula/ BillingDetails.`

java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@org.hibernate.annotations.DiscriminatorFormula(
    "case when CARDNUMBER is not null then 'CC' else 'BA' end"
)
public abstract class BillingDetails {
    // ...
}
```

该架构中没有识别器列，所以映射依赖 SQL CASE/WHEN 表达式来判定特定行是表示一张信用卡还是一个银行账户(许多开发人员都从未使用过此类 SQL 表达式；如果不熟悉它，可以查看 ANSI 标准)。该表达式的结果是文字、CC 或 BA，这是在子类映射上声明的。

每个类层次结构一个表的策略的缺点可能对于你的设计来说会太过严重——思考一下，从长期来看，反规范化的架构会变成沉重负担。你的 DBA 可能完全不喜欢它。下一个继承关系映射策略将让你避免此问题。

6.4 每个带有联结的子类使用一个表

第四个选项是将继承关系表示为 SQL 外键关联。声明持久化属性的每一个类/子类——包括抽象类甚至接口——都有其自己的表。

不同于一开始映射的每个具体类一个表的策略，此处具体 `@Entity` 的表包含仅用于每个非继承属性的列(由子类本身声明)，以及一个主键(它也是超类表的一个外键)。这实际上要比看起来的容易，如图 6-3 所示。

如果让 `CreditCard` 子类的实例持久化，则 Hibernate 会插入两行。由 `BillingDetails` 超类声明的属性值会被存储到 `BILLINGDETAILS` 表的一个新行中。只有由子类声明的属性值会被存储到 `CREDITCARD` 表的新行中。被这两行所共享的主键会将它们连接在一起。之后，可以通过联结子类表和超类表来从数据库中检索该子类实例。

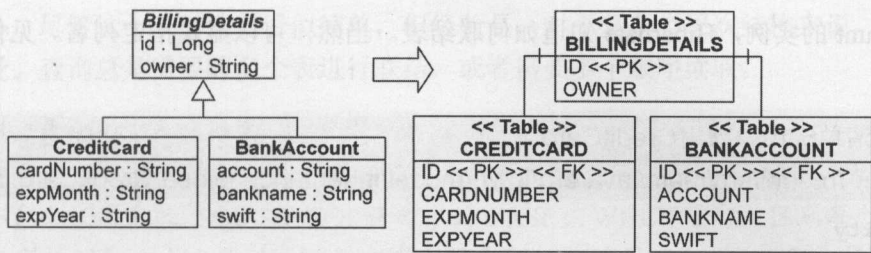


图 6-3 将层次结构的所有类映射到其自己的表

该策略的主要优势是，它规范化 SQL 架构。架构演化和完整性约束定义简单明了。引用特定子类的表的外键可以表示对该特定子类的多态关联。使用 JOINED 继承关系策略来创建每个子类层次结构一个表的映射，见代码清单 6.8。

代码清单6.8 使用JOINED映射BillingDetails

路径: /model/src/main/java/org/jpwh/model/inheritance/joined/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String owner;
    // ...
}
```

根类 BillingDetails 被映射到表 BILLINGDETAILS。注意，使用这个策略不需要识别器。在子类中，如果子类表的主键列有(或应该有)与超类表主键列相同的名称，则就不需要指定联结列。见代码清单 6.9。

代码清单6.9 映射BankAccount(具体类)

路径: /model/src/main/java/org/jpwh/model/inheritance/joined/BankAccount.java

```
@Entity
public class BankAccount extends BillingDetails {

    @NotNull
    protected String account;

    @NotNull
    protected String bankname;

    @NotNull
    protected String swift;

    // ...
}
```

这个实体没有标识符属性；它会自动继承 ID 属性和来自超类的列，而如果希望检索

BankAccount 的实例，Hibernate 知道如何联结表。当然，可以显式指定列名。见代码清单 6.10。

代码清单6.10 映射CreditCard

路径: /model/src/main/java/org/jpwh/model/inheritance/joined/CreditCard.java

```
@Entity
@PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
public class CreditCard extends BillingDetails {

    @NotNull
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;
    // ...
}
```

BANKACCOUNT 和 CREDITCARD 表的主键列也都具有引用 BILLINGDETAILS 表主键的外键约束。

为了执行 `select bd from BillingDetailsbd`，Hibernate 依赖于 SQL 外联结：

```
select
    BD.ID, BD.OWNER,
    CC.EXPMONTH, CC.EXPYEAR, CC.CARDNUMBER,
    BA.ACCOUNT, BA.BANKNAME, BA.SWIFT,
    case
        when CC.CREDITCARD_ID is not null then 1
        when BA.ID is not null then 2
        when BD.ID is not null then 0
    end
from
    BILLINGDETAILS BD
    left outer join CREDITCARD CC on BD.ID=CC.CREDITCARD_ID
    left outer join BANKACCOUNT BA on BD.ID=BA.ID
```

SQL CASE ... WHEN 子句检测子类表 CREDITCARD 和 BANKACCOUNT 中行的存在(或缺失)，这样 Hibernate 就能判定用于 BILLINGDETAILS 表特定行的具体子类。

对于像 `select cc from CreditCard cc` 这样的小范围子类查询，Hibernate 会使用内联结：

```
select
    CREDITCARD_ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
from
    CREDITCARD
    inner join BILLINGDETAILS on CREDITCARD_ID=ID
```

正如你所看到的，要手动实现这个映射策略会更加困难——甚至专门的报告也会更复杂。如果计划用手动编写的 SQL 混合 Hibernate 代码，这就是一个重要的考虑项。

此外，尽管这个映射策略看似简单，但经验是，对于复杂类层次结构来说，其性能将不可接受。查询总是需要跨多个表进行联结，或者需要多个顺序读取。

带有联结和识别器的继承

Hibernate 不需要特殊的识别器数据库列来实现 InheritanceType.JOINED 策略，且 JPA 规范也不包含任何需求。SQL SELECT 语句中的 CASE ... WHEN 子句是区别每个被检索行的实体类型的明智方式。不过，可能会在别处找到一些使用 InheritanceType.JOINED 和 @DiscriminatorColumn 映射的 JPA 示例。显然，有些 JPA 提供程序不使用 CASE ... WHEN 子句，而是依赖识别器值，即使是用于 InheritanceType.JOINED 策略也是如此。Hibernate 不需要识别器，而是使用一个声明的 @DiscriminatorColumn，即使是使用 JOINED 映射策略。如果忽略带有 JOINED 的识别器映射(它在较老的 Hibernate 版本中会被忽略)，则可以启用配置属性 hibernate.discriminator.ignore_explicit_for_joined。

在介绍何时选择哪种策略之前，先来考虑在单个类层次结构中混合继承关系映射策略的情况。

6.5 混合继承策略

可以使用 TABLE_PER_CLASS、SINGLE_TABLE 或 JOINED 策略映射整个继承层次结构。不能混合它们——例如，从带有识别器的每个类层次结构一个表切换到规范的每个子类一个表的策略。一旦选择了继承策略，就必须坚持使用它。

不过，这并非完全正确。通过使用一些技巧，就可以为特定子类切换映射策略。例如，可以将类层次结构映射到单个表，但对于特定子类，则可以使用外键映射策略切换到单独表，就像每个子类一个表一样。看看图 6-4 中的架构。

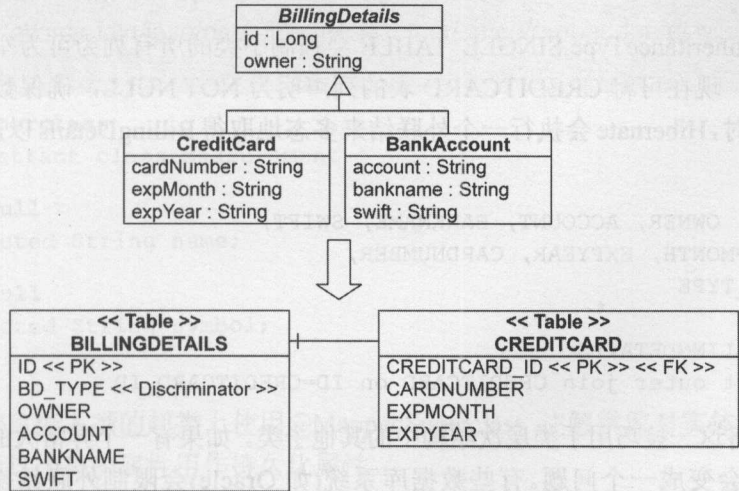


图 6-4 将一个子类引导到其专有的辅助表

使用 InheritanceType.SINGLE_TABLE 映射超类 BillingDetails，就像之前所做的那样。现在映射希望从单个表引导到辅助表的子类。见代码清单 6.11。

代码清单6.11 映射CreditCard

路径: /model/src/main/java/org/jpwh/model/inheritance/mixed/CreditCard.java

```

@Entity
@DiscriminatorValue("CC")
@SecondaryTable(
    name = "CREDITCARD",
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
)
public class CreditCard extends BillingDetails {

    @NotNull ← 为 DDL 而被 JPA 忽略; 策略是 SINGLE_TABLE
    @Column(table = "CREDITCARD", nullable = false) ← 重写该主表
    protected String cardNumber;

    @Column(table = "CREDITCARD", nullable = false)
    protected String expMonth;

    @Column(table = "CREDITCARD", nullable = false)
    protected String expYear;
    // ...
}

```

`@SecondaryTable` 和 `@Column` 注解会分组一些属性并告知 Hibernate 从辅助表中获取它们。要使用该辅助表的名称来映射所有移动到辅助表中的属性。这是使用 `@Column` 的表 (table) 参数完成的, 此前没有介绍过它。这个映射有许多用途, 稍后将再次对其进行介绍。在这个示例中, 它会将 `CreditCard` 属性从单个表策略分离到 `CREDITCARD` 表中。

这个表的 `CREDITCARD_ID` 列同时也是主键, 且它有一个引用单个层次结构表 ID 的外键约束。如果不为辅助表指定一个主键联结列, 则会使用单个继承关系表主键的名称——本例中是 `ID`。

记住, `InheritanceType.SINGLE_TABLE` 会强制子类的所有列为可为空。这个映射的一个好处在于, 现在可将 `CREDITCARD` 表的列声明为 `NOT NULL`, 确保数据完整性。

在运行时, Hibernate 会执行一个外联结来多态地取得 `BillingDetails` 以及所有子类实例:

```

select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT,
    EXPMONTH, EXPYEAR, CARDNUMBER,
    BD_TYPE
from
    BILLINGDETAILS
    left outer join CREDITCARD on ID=CREDITCARD_ID

```

还可以将这一技巧用于类层次结构中的其他子类。如果有一个异常大的类层次结构, 那么外联结就会变成一个问题。有些数据库系统(如 Oracle)会限制外联结操作中表的数量。对于一个大的层次结构, 可能希望切换到另一种提取策略, 它要能执行即时的第二个 SQL 查询而非外联结。

Hibernate 特性

在撰写本书时，JPA 或 Hibernate 注解中还没有可用于这一映射的提取策略的切换，因此必须在原生 Hibernate XML 文件中映射这个类：

路径：/model/src/main/resources/inheritance/mixed/FetchSelect.hbm.xml

```
<subclass name="CreditCard"
    discriminator-value="CC">
    <join table="CREDITCARD" fetch="select">
    ...
    </join>
</subclass>
```

到目前为止，仅探讨了实体继承关系。尽管 JPA 规范不清楚@Embeddable 类的继承关系和多态性，但 Hibernate 提供了用于组件类型的映射策略。

6.6 可嵌入类的继承

可嵌入类是其所属实体的一个组件；因此，不会应用本章中所介绍过的常规实体继承规则。作为 Hibernate 的扩展，可以映射一个继承来自超类(或接口)的一些持久化属性的可嵌入类。思考一个拍卖商品的这两个新属性：尺寸和重量。

一个商品的尺寸就是其宽度、高度和深度，用指定单位及其符号表示：例如，英寸(")或厘米(cm)。商品的重量也具有度量单位，如磅(lbs)或千克(kg)。要捕获度量的常用属性(名称和符号)，则要定义用于 Dimension 和 Weight 的一个名为 Measurement 的超类。见代码清单 6.12。

代码清单6.12 映射Measurement抽象可嵌入超类

路径：/model/src/main/java/org/jpwh/model/inheritance/embeddable/ Measurement.
java

```
@MappedSuperclass
public abstract class Measurement {

    @NotNull
    protected String name;

    @NotNull
    protected String symbol;
    // ...
}
```

在映射的该可嵌入类的超类上使用@MappedSuperclass 注解就像对实体所做的那样即可。子类将继承这个类的属性用作持久化属性。

要将 Dimensions 和 Weight 子类定义为@Embeddable。对于 Dimensions，需要重写所有超类属性并添加一个列名前缀。见代码清单 6.13。

代码清单6.13 映射Dimensions类

路径: /model/src/main/java/org/jpwh/model/inheritance/embeddable/Dimensions.java

```
@Embeddable
@AttributeOverrides({
    @AttributeOverride(name = "name",
        column = @Column(name = "DIMENSIONS_NAME")),
    @AttributeOverride(name = "symbol",
        column = @Column(name = "DIMENSIONS_SYMBOL"))
})
public class Dimensions extends Measurement {

    @NotNull
    protectedBigDecimal depth;

    @NotNull
    protectedBigDecimal height;

    @NotNull
    protectedBigDecimal width;
    // ...
}
```

不使用该重写,同时嵌入 Dimension 和 Weight 的 Item 就会映射到具有冲突列名称的表。以下是 Weight 类;其映射也会使用一个前缀重写列名称(为了保持一致,使用前面的重写避免了冲突)。见代码清单 6.14。

代码清单6.14 映射Weight类

路径: /model/src/main/java/org/jpwh/model/inheritance/embeddable/Weight.java

```
@Embeddable
@AttributeOverrides({
    @AttributeOverride(name = "name",
        column = @Column(name = "WEIGHT_NAME")),
    @AttributeOverride(name = "symbol",
        column = @Column(name = "WEIGHT_SYMBOL"))
})
public class Weight extends Measurement {
    @NotNull
    @Column(name = "WEIGHT")
    protectedBigDecimal value;
    // ...
}
```

所属实体 Item 会定义两个常规持久化嵌入属性。见代码清单 6.15。

代码清单6.15 映射Item类

路径: /model/src/main/java/org/jpwh/model/inheritance/embeddable/Item.java

```
@Entity
```



```
public class Item {  
    protected Dimensions dimensions;  
    protected Weight weight;  
    // ...  
}
```

图 6-5 揭示了这一映射。也可以重写 Item 类中可嵌入属性的冲突 Measurement 列名称，如 5.2 节所述。相反，更愿意只重写它们一次，在 @Embeddable 类中，所以这些类的所有使用者都不必解决这个冲突。

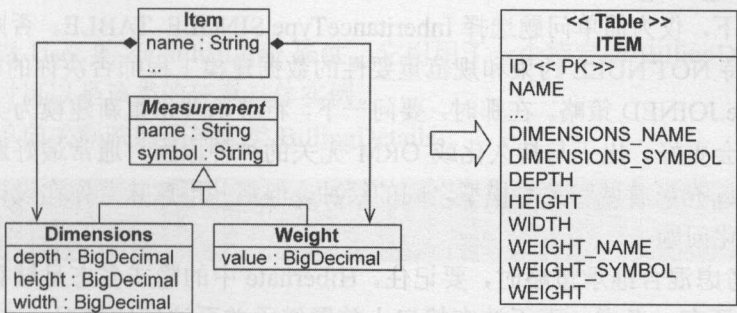


图 6-5 使用具体可嵌入类的继承属性映射它们

要当心的一个陷阱是在实体(如 Item)中嵌入抽象超类类型(如 Measurement)的一个属性。这永远无法运行；JPA 提供程序不清楚如何多态存储和加载 Measurement 实例。它没有决定数据库中的值是 Dimension 还是 Weight 实例的信息，因为没有识别器。这意味着尽管可以使用一个 @Embeddable 类从 @MappedSuperclass 继承一些持久化属性，但对实例的引用并非多态的——它总是会命名一个具体类。

将其与 5.3.2 小节“2. 转换组件的属性”中介绍的用于可嵌入类的可选继承关系策略作比较，该策略支持多态但需要一些自定义的类型识别代码。

接下来，会介绍更多与如何为应用程序类层次结构选择映射策略的合适组合有关的更多小技巧。

6.7 选择一种策略

选择合适的继承-映射策略取决于实体层次结构超类的用途。必须考虑查询超类实例的频率以及是否有针对超类的关联。另一个重要方面是超类型和子类型的属性：子类型是否有许多附加属性或者只有不同于其超类型的行为。下面是一些经验法则：

- 如果不需要多态关联或查询，则可以倾向于每个具体类一个表——换句话说，如果从不或很少 select bd from BillingDetailsbd 并且没有类与 BillingDetails 关联，就可以这么做。应该选择一个使用 InheritanceType.TABLE_PER_CLASS 的基于 UNION 的显式映射，因为之后(优化过的)多态查询和关联将会可行。

- 如果确实需要多态关联(到超类的关联, 即在运行时关联到具有具体类动态解析的层次结构中的所有类)或查询, 并且子类声明了相对较少的属性(尤其是子类之间的主要区别是在其行为中时), 则要倾向于 `InheritanceType.SINGLE_TABLE`。你的目标是最小化可为空的列的数量, 以及让你自己(和 DBA)相信, 反规范化架构在长期的运行中不会造成问题。
- 如果确实需要多态关联或查询, 并且子类声明了许多(非可选)属性(主要通过子类持有的数据来区分它们), 则要倾向于 `InheritanceType.JOINED`。或者, 根据继承关系层次结构的广度和深度以及联结对比联合的可能开销, 而使用 `InheritanceType.TABLE_PER_CLASS`。这一决策可能需要使用真实数据对 SQL 执行计划进行评估。

默认情况下, 仅为简单问题选择 `InheritanceType.SINGLE_TABLE`。否则, 对于复杂情况, 或者在坚持 NOT NULL 约束和规范重要性的数据建模工程师否决你的时候, 应该考虑 `InheritanceType.JOINED` 策略。在那时, 要问一下, 将继承关系重新建模为类模型中的委托是否可能并不会更好。出于与持久化或 ORM 无关的种种原因, 通常最好避免复杂的继承关系。Hibernate 充当着域和关系模型之间的缓冲, 但那并不意味着你在设计类的时候可以完全忽略持久化问题。

当开始考虑混合继承策略时, 要记住, Hibernate 中的隐式多态足够智能, 足以处理外来的情况。还有, 考虑一下无法在接口上放置继承关系注解的情况; 这在 JPA 中并非标准的。

例如, 思考一下示例应用程序中的一个额外接口: `ElectronicPaymentOption`。这是一个业务接口, 它没有持久化方面——只不过在该应用程序中, 像 `CreditCard` 这样的持久化类很可能要实现这一接口。无论如何映射 `BillingDetails` 层次结构, Hibernate 都可以正确响应查询 `select o from ElectronicPaymentOption o`。如果其他并非 `BillingDetails` 层次结构的一部分的类被映射为持久化并且实现这一接口, 这甚至也是可行的。Hibernate 总是清楚要查询哪些表、要构造哪些实例、以及如何返回一个多态结果。

可以将所有的映射策略应用到抽象类。即便你查询或加载抽象类, Hibernate 也不会尝试实例化它。

几次提到过 `User` 和 `BillingDetails` 之间的 6.8 节关系以及它如何影响继承关系映射策略的选择。将详细通过更高级的主题来探究它: 多态关联。如果模型现在没有这样的关系, 那么可能希望稍后当在应用程序中遇到该问题时再阅读这些内容。

6.8 多态关联

多态是 Java 等面向对象语言的一个本质特性。支持多态关联和多态查询是像 Hibernate 这样的 ORM 解决方案的基本功能。出乎意料的是, 在无须过多讨论多态的情况下已经设法走到这一步了。令人振奋的是, 关于该主题没有太多可以说的——在 Hibernate 中使用多态如此简单, 无须花费太多精力来解释它。

为了提供概况, 首先考虑对一个可能有多个子类的类进行多对一关联, 然后是一对多关系。对于这两个示例, 域模型的类都是相同的, 如图 6-6 所示。

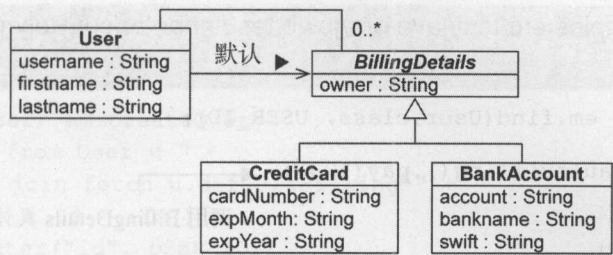


图 6-6 将信用卡或者银行账户作为默认结算详情的用户

6.8.1 多态多对一关联

首先，考虑 User 的 defaultBilling 属性。它引用了一个特定的 BillingDetails 实例，这个实例在运行时可以是该类的任意具体实例。

要将这个单向关联映射到抽象类 BillingDetails:

路径: /model/src/main/java/org/jpwh/model/inheritance/associations/manytoone/
User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @ManyToOne(fetch = FetchType.LAZY)
    protected BillingDetails defaultBilling;
    // ...
}
```

现在 USERS 表具有表示这一关系的联结/外键列 DEFAULTBILLING_ID。它是一个可为空的列，因为 User 可能没有指定的默认结算方法。由于 BillingDetails 是抽象的，所以该关联必须在运行时引用 BillingDetails 其中一个子类的实例——CreditCard 或 BankAccount。

不必做任何特殊的事情来启用 Hibernate 中的多态关联; 如果一个关联的目标类是使用 @Entity 和 @Inheritance 来映射的，则该关联自然而然就是多态的。

以下代码揭示了对 CreditCard 子类实例的关联的创建:

路径: /examples/src/test/java/org/jpwh/test/inheritance/ PolymorphicManyToOne.
java

```
CreditCard cc = new CreditCard(
    "John Doe", "1234123412341234", "06", "2015"
);
User johndoe = new User("johndoe");
johndoe.setDefaultBilling(cc);

em.persist(cc);
em.persist(johndoe);
```

现在，当在第二个工作单元中导航到该关联时，Hibernate 会自动检索 CreditCard 实例:

路径: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
User user = em.find(User.class, USER_ID);
user.getDefaultBilling().pay(123);
```

调用 BillingDetails 具体子类上的 pay() 方法

只有一件事情需要密切留意: 由于 defaultBilling 属性是使用 FetchType.LAZY 来映射的, 所以Hibernate 会代理该关联目标。这种情况下, 就不能在运行时对 CreditCard 这个具体类执行类型转换, 甚至 instanceof 操作符都会有不寻常的行为:

路径: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
User user = em.find(User.class, USER_ID);
BillingDetails bd = user.getDefaultBilling();
assertFalse(bd instanceof CreditCard);    别这么做——ClassCastException!
// CreditCard creditCard = (CreditCard) bd;
```

在这个例子中, Bd 引用不是 CreditCard 实例; 它是运行时生成的 BillingDetails 的特殊子类, 即 Hibernate 代理。当调用该代理上的方法时, Hibernate 会将该调用委托给它延迟得到的 CreditCard 的实例。直到这一实例化发生之前, Hibernate 都不清楚指定实例的子类型是什么——这可能需要访问一次数据库, 而这正是一开始使用延迟加载所试图避免的。要执行代理安全的类型转换, 可以使用 em.getReference():

路径: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
User user = em.find(User.class, USER_ID);
BillingDetails bd = user.getDefaultBilling();
CreditCard creditCard =
    em.getReference(CreditCard.class, bd.getId());
assertTrue(bd != creditCard);
```

没有 SELECT

当心!

在 getReference()调用之后, bd 和 creditCard 会引用两个不同的代理实例, 这两个代理实例都委托到相同的基础 CreditCard 实例。不过, 第二个代理有不同的接口, 并且可以调用像 creditCard.getExpMonth()这样仅应用到该接口的方法(注意, 如果使用字段访问映射该 id 属性, bd.getId()会触发 SELECT)。

可以通过避免延迟获取来规避这些问题, 如下所示使用一个即刻提取查询:

路径: /examples/src/test/java/org/jpwh/test/inheritance/ PolymorphicManyToOne.java

```
User user = (User) em.createQuery(
    "select u from User u " +
    "left join fetch u.defaultBilling " +
    "where u.id = :id")
    .setParameter("id", USER_ID)
    .getSingleResult();
```

还没有使用代理:
BillingDetails 实例即
刻提取了

```
CreditCard creditCard = (CreditCard) user.getDefaultBilling(); ←
```

真正面向对象的代码不应该使用 `instanceof` 或大量的类型转换。如果发现使用代理出现了问题,则应该质疑路径设计,想想是否有一种更为多态的方法。Hibernate 还提供了字节码方式作为通过代理延迟加载的可选项;第12章将我们回到提取策略的探讨。

可以用相同的方式处理一对一关联。那么,像用于每个 User 的 `billingDetails` 集合这样的多元关联又该怎么办呢?

6.8.2 多态集合

一个 User 可能有对多个 `BillingDetails` 的引用,而不是只有一个默认的(其中一个就是默认的;暂时忽略它)。可以使用双向一对多关联映射它:

路径: /model/src/main/java/org/jpwh/model/inheritance/associations/onetomany/ User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @OneToMany(mappedBy = "user")
    protected Set<BillingDetails> billingDetails = new HashSet<>();
    // ...
}
```

接着,这里是该关系的所属部分(在上一个映射中使用 `mappedBy` 声明的):

路径: /model/src/main/java/org/jpwh/model/inheritance/associations/onetomany/ BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {

    @ManyToOne(fetch = FetchType.LAZY)
    protected User user;
    // ...
}
```

到目前为止,关于这一关联映射并没有什么特殊事项。`BillingDetails` 类层次结构可以

使用 `TABLE_PER_CLASS`、`SINGLE_TABLE` 或一个 `JOINED` 继承类型来映射。在加载该集合元素时，Hibernate 足够智能，知道使用带有 `JOIN` 或 `UNION` 操作符的正确 SQL 查询。

不过，这里有一个限制：`BillingDetails` 类不能是 `@MappedSuperclass`，就像第 6.1 节中所介绍的那样。它必须使用 `@Entity` 和 `@Inheritance` 来映射。

具有隐式多态的关联

Hibernate 提供了一种“终极手段”技术，如果真的必须在不使用 `@Inheritance` 显式映射类层次结构的情况下将关联映射到该类层次结构，就可以使用该技术。只要使用 Hibernate 原生 XML 映射和 `<any/>` 元素即可。如果需要，建议在 Hibernate 文档或者本书的上一个版本中查阅它，但只要可能，就应该尝试避免它，因为它会产生很难看明白的架构。

6.9 本章小结

- 具有隐式多态的每个具体类使用一个表是映射实体继承关系层次结构的最简单策略，但它不能很好地支持多态关联。此外，不同表的不同列会完全共享相同的语义，让架构演化更加复杂。建议仅将这个方案用于类层次结构的顶层，其中多态通常并不需要并且未来也不太可能出现修改超类的时候。
- 使用联合的每个具体类一个表的策略是可选的，且 JPA 实现可能不支持它，但它确实能处理多态关联。
- 每个类层次结构一个表的策略在性能和简单性方面都是最佳的：没有复杂联结或联合的专用报告是可以的，且架构演化也简单明了。一个重要的问题是数据完整性，因为你必须将一些列声明为可为空。另一个问题是规范化：这一策略会创建非键列之间的函数依赖，这违反了第三范式。
- 使用联结的每个子类一个表的策略的主要优势在于，它规范了 SQL 架构，让架构演化和完整性约束定义简单明了。其劣势在于，它比手动实现更加困难，且对于复杂类层次结构来说，其性能可能无法接受。

映射集合和实体关联

第 7 章

7



本章内容简介：

- 映射持久化集合
- 基本和可嵌入类型的集合
- 简单多对一和一对多实体关联

根据对于 Hibernate 用户社区的经验，当许多开发人员开始使用 Hibernate 的时候，想做的第一件事情就是映射父/子关系。这通常是他们首次遇到集合。也是他们第一次必须考虑实体和值类型之间的区别，否则就会迷失在 ORM 的复杂性之中。

管理类之间的关联以及表之间的关系是 ORM 的核心。实现 ORM 解决方案所涉及的大多数难题都与集合和实体关联管理有关。可以随时回顾本章，以便完全领会这个主题。在本章开头会介绍基本集合映射概念以及简单示例。之后，将准备好处理实体关联中的第一个集合——尽管第 8 章会介绍更为复杂的实体关联映射。为了有一个全局概念，建议阅读完本章和第 8 章。

JPA 2 中主要的新功能

- 支持集合以及基本和可嵌入类型的映射。
- 支持在附加数据库列中存储每个元素索引的持久化列表。
- 一对多关联现在具有一个孤儿删除选项。

7.1 集、包、列表及值类型映射

Java 有一个富集合 API，可以从中选择最适合域模型设计的接口和实现。查看最常见的集合映射，用最小的变化来重复相同的 Image 和 Item 示例。通常首先要查看数据库架构并且创建和映射集合属性。然后要继续介绍如何选择一个特定集合接口以及映射各种集合类型：集、标识符包、列表、映射，以及最后排列过和排序过的集合。

7.1.1 数据库架构

下面扩展 CaveatEmptor 并支持将图片附加到拍卖商品。暂时先忽略 Java 代码，并回过头来仅考虑数据库架构。

数据库中需要一个 IMAGE 表来保存这些图片，或者可能只保存图片的文件名。这个表还有一个外键列，如 ITEM_ID，它会引用 ITEM 表。看一下图 7-1 中所示的架构。

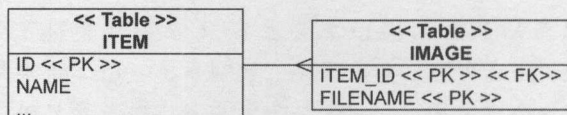


图 7-1 IMAGE 表保存图片文件名，每个文件名引用一个 ITEM_ID

这就是该架构的全部了——没有集合或组合生命周期(可以在 ITEM_ID 外键列上使用一个 ON DELETE CASCADE 选项。当应用程序删除一个 ITEM 行时，数据库会自动删除数据库中引用这个 ITEM 的 IMAGE 行。暂时假定情况并非如此)。

7.1.2 创建和映射一个集合属性

如何用目前所掌握的知识映射这个 IMAGE 表？大概会将其映射成名称为 Image 的 @Entity 类。在本章稍后的内容中，要使用 @ManyToOne 属性映射一个外键列。还需要一个组合主键映射用于该实体类，如 9.2.2 小节中所示。

其中没有映射的集合；它们不是必须的。当需要一个商品的图片时，可以用 JPA 查询语言编写并执行一个查询：select img from Image img where img.item = :itemParameter。持久化集合总是可选功能。为何要映射集合？

可以创建的集合是 Item#images，为特定商品引用所有的图片。要创建并映射这个集合属性来完成以下任务：

- 在调用 someItem.getImages() 时自动执行 SQL 查询 SELECT * from IMAGE where ITEM_ID = ?。只要域模型实例处于托管状态(之后)，就可以根据需要在导航类之间的关联时从数据库进行读取。不必使用 EntityManager 手动编写和执行一个查询来加载数据。另一方面，在迭代该集合时，集合查询就一直会是“此商品的所有图片”，而绝不是“只匹配条件 XYZ 的图片”。
- 要避免使用 entityManager.persist() 保存每个 Image。如果有一个映射集合，那么使用 someItem.getImages().add() 将 Image 添加到集合就会在保存 Item 时让其自动持久化。这一级联持久化很方便，因为可以在不调用 EntityManager 的情况下保存实例。
- 让 Images 使用依赖性的生命周期。当删除一个 Item 时，Hibernate 会使用额外的 SQL DELETE 删除所有附加的 Images。不必担心图片的生命周期以及清除孤立对象(假定数据库外键约束未处于 ON DELETE CASCADE)。JPA 提供程序会处理组合生命周期。

重要的是要认识到，尽管这些好处听起来很棒，但你要付出的代价就是额外的映射复杂性。我们见到过许多 JPA 初学者纠结于集合映射，而“为何这样做？”的答案经常是“我

认为这个集合是要求用的。”

分析用于拍卖商品的图片的场景，你从集合映射中获得了好处。图片有了依赖生命周期；删除一个商品时，所有附加的图片都应该被删除。存储商品时，所有附加的图片都应该被存储。并且当展示一个商品时，通常还要展示所有的图片，因此 `someItem.getImages()` 在 UI 代码中是很方便的。不必再次调用持久化服务来得到图片；它们就在手边。

现在，开始选择最适合域模型设计的集合接口和实现。下面看看最常见的集合映射，用最小的变化来重复相同的 `Image` 和 `Item` 示例。

7.1.3 选择集合接口

Java 域模型中集合属性的习惯用语如下：

```
<<Interface>> images = new <<Implementation>>();  
  
// Getter and setter methods  
// ...
```

使用一个接口来声明属性的类型，而非实现。选取一种匹配实现，并且立即初始化该集合；这样做可以避免有未初始化的集合。不建议延迟构造函数或设置方法中的初始化集合。

使用泛型，下面是一个典型的 `Set`：

```
Set<String> images = new HashSet<String>();
```

不使用泛型的原始集合

如果不使用泛型或映射的键/值类型指定集合元素的类型，就需要将类型告知 `Hibernate`。例如，不使用 `Set<String>`，而是使用 `@ElementCollection(targetClass=String.class)` 来映射一个原始 `Set`。这也适用于 `Map` 的类型参数。使用 `@MapKeyClass` 指定 `Map` 的键类型。本书中的所有示例都使用泛型集合及映射，你也应该如此。

`Hibernate` 支持开箱即用的最重要 `JDK` 集合接口并且以持久化方式保留了 `JDK` 集合、映射和数组的语意。每个 `JDK` 接口都有 `Hibernate` 所支持的一个匹配实现，并且使用正确的组合很重要。`Hibernate` 会包装已经在字段声明上初始化过的集合，或者在它不正确时替换它。此外，它那样做是为了启用对集合元素的延迟加载和脏检查。

如果不扩展 `Hibernate`，可以从以下集合中选择：

- `java.util.Set` 属性，它是使用 `java.util.HashSet` 初始化的。不会保存元素的顺序，且不允许重复元素。所有的 JPA 提供程序都支持这个类型。
- `java.util.SortedSet` 属性，它是使用 `java.util.TreeSet` 来初始化的。这个集合支持元素的固定顺序：排列发生在内存中，在 `Hibernate` 加载数据之后进行。这是仅用于 `Hibernate` 的扩展；其他 JPA 提供程序可能会忽略集的“已排列”方面。
- `java.util.List` 属性，它是使用 `java.util.ArrayList` 来初始化的。`Hibernate` 会在数据库表中使用额外的索引列保存每个元素的位置。所有的 JPA 提供程序都支持这个类型。

- `java.util.Collection` 属性,它是使用 `java.util.ArrayList` 来初始化的。这个集合具有包(bag)语意;可以重复,但不保存元素的顺序。所有的 JPA 提供程序都支持这个类型。
- `java.util.Map` 属性,它是使用 `java.util.HashMap` 来初始化的。可以在数据库中保存映射的键值对。所有的 JPA 提供程序都支持这个类型。
- `java.util.SortedMap` 属性,它是使用 `java.util.TreeMap` 来初始化的。它支持元素的固定顺序:排列发生在内存中,在 Hibernate 加载数据之后进行。这是仅用于 Hibernate 的扩展;其他 JPA 提供程序可能会忽略映射的“已排列”方面。
- Hibernate 支持持久化数组,但 JPA 不支持。它们很少用到,本书也不会介绍:Hibernate 不能包装数组属性,因此集合的许多优势无法发挥作用,如按需要延迟加载。如果确定无须延迟加载,则可以在域模型中只使用持久化数组(可以按需要加载数组,但这需要使用字节码增强来拦截,12.1.3 节将会介绍)。

Hibernate 特性

如果希望映射 Hibernate 不直接支持的集合接口和实现,则需要告知 Hibernate 自定义集合的语意。Hibernate 中的扩展点是 `org.hibernate.collection.spi` 包中的 `PersistentCollection` 接口,通常可以在其中扩展已有的 `PersistentSet`、`PersistentBag` 和 `PersistentList` 类之一。编写自定义持久化集合并不容易,并且如果不是有经验的 Hibernate 用户,不建议这么做。可以在 Hibernate 测试套件的源代码中找到一个示例。

对于拍卖商品和图片示例来说,假定图片存储在文件系统的某个位置,而你仅将文件名保存在数据库中。

事务性文件系统

如果仅在 SQL 数据库中保存图片的文件名,则必须在某个位置存储每个图片的二进制数据——图片文件。可以在 SQL 数据库的 BLOB 列中存储图片数据(参阅 5.3.1 小节“4. 二进制和大型值类型”部分)。如果决定不在数据库中存储图片,而是存储常规文件,则应该注意,标准 Java 文件系统 API `java.io.File` 和 `java.nio.file.Files` 不是事务型的。文件系统操作并未列入(JTA)系统事务中;使用 Hibernate 将文件名写入 SQL 数据库中可能可以成功完成一个事务,但之后在文件系统中存储或删除该文件可能会失败。这些操作无法像原子单元那样回滚,并且无法正确隔离操作。

幸运的是,有现成的 Java 开源事务性文件系统实现可用,如 XADisk(参见 <https://xadisk.java.net>)。可以轻松地将 XADisk 与系统事务管理器集成起来,如本书示例用到的 Bitronix。之后文件操作会在相同的 `UserTransaction` 中与 Hibernate 的 SQL 操作一起被列入、提交和回滚。

映射 Item 图片文件名的集合。

7.1.4 映射集

最简单的实现是 `String` 图片文件名的 `Set`(集)。将集合属性添加到 `Item` 类,如代码清单 7.1 所示。

代码清单7.1 被映射为简单字符串集的图片

路径: /model/src/main/java/org/jpwh/model/collections/setofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(
        name = "IMAGE",
        joinColumns = @JoinColumn(name = "ITEM_ID"))
    @Column(name = "FILENAME")
    protected Set<String> images = new
        HashSet<String>();
    // ...
}
```

默认为 ITEM_IMAGES

默认

默认为 IMAGES

在这里初始化字段

需要将 JPA 注解@ElementCollection 用于值类型元素的集合。如果没有@CollectionTable 和@Column 注解，Hibernate 会使用默认的架构名称。看一下图 7-2 中的架构：下划线标记出了主键列。

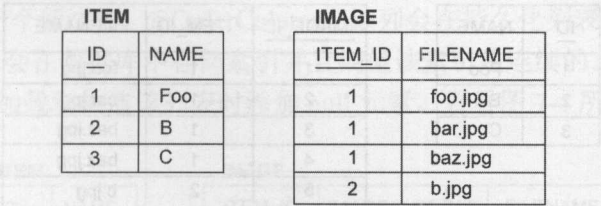


图 7-2 用于字符串集的表结构和示例数据

IMAGE 表有 ITEM_ID 和 FILENAME 列的组合主键。那意味着不能有重复行：每个图片文件只能附加到一个商品一次。不会存储图片的顺序。这适合于域模型和 Set 集合。看起来不大可能允许用户将相同的图片多次附加到相同商品，但假定这么做了，那么哪种映射是合适的呢？

Hibernate 特性

7.1.5 映射标识符包

包(bag)是允许重复元素的未排序集合，就像 java.util.Collection 接口。奇怪的是，Java Collections 框架未包含包实现。用 ArrayList 初始化该属性，而 Hibernate 会在存储和加载元素时忽略元素的索引。见代码清单 7.2。

代码清单7.2 字符串的包，允许重复元素

路径: /model/src/main/java/org/jpwh/model/collections/bagofstrings/Item.java

```
@Entity
public class Item {
```

```
@ElementCollection
@CollectionTable(name = "IMAGE")    代理主键允许重复
@Column(name = "FILENAME")
@org.hibernate.annotations.CollectionId(
    columns = @Column(name = "IMAGE_ID"),
    type = @org.hibernate.annotations.Type(type = "long"),
    generator = Constants.ID_GENERATOR)
protected Collection<String> images = new ArrayList<String>();
// ...
}
```

JDK 中没有

BagImpl

①代理主键列

②仅适用于 Hibernate 的注解

③配置主键

这看起来更为复杂：不能像之前那样继续使用相同的架构。IMAGE 集合表需要不同的主键来允许用于每个 ITEM_ID 的重复 FILENAME 值。引入名为 IMAGE_ID 的代理主键列①，并且使用仅适用于 Hibernate 的注解②来配置生成该主键的方式③。如果忘记了键生成器，参阅 4.2.4 小节。图 7-3 中显示了修改后的架构。

ITEM	
ID	NAME
1	Foo
2	B
3	C

IMAGE		
IMAGE_ID	ITEM_ID	FILENAME
1	1	foo.jpg
2	1	bar.jpg
3	1	baz.jpg
4	1	baz.jpg
5	2	b.jpg

图 7-3 用于字符串包的代理主键列

这里有一个有意思的问题：如果看到的完全是这个架构，那么你能弄明白如何在 Java 中映射表吗？ITEM 和 IMAGE 表看起来一样：每个表都有代理主键列以及其他一些规范化列。每个表都可以使用@Entity 类来映射。不过，即便会存在组合生命周期，我们还是决定使用 JPA 功能并将集合映射到 IMAGE。实际上，这就是一个决定而已，对于这个表来说，所需要的就是一些预定义查询和操作规则，而非更为通用的@Entity 映射。做出这样的决定时，确保清楚原因和后果。

接下来的映射技术会使用列表来保存图片顺序。

7.1.6 映射列表

还没有使用 ORM 软件之前，持久化列表看起来是非常强大的概念；想象一下，用普通的 JDBC 和 SQL 存储和加载 java.util.List<String>需要多少工作量。如果将元素添加到该列表的中间，根据列表的实现，该列表就会将后续所有元素变动到正确的指针或重新排列指针。如果从列表的中间移除一个元素，则会发生别的变化，依此类推。如果 ORM 软件可以为数据库记录自动完成所有这些工作，就会使持久化列表看起来比其实际上更具吸引力。

正如 3.2.4 小节中介绍的，第一反应通常是按照用户输入数据元素的顺序来保存它们。之后通常必须以相同顺序显示它们。但如果可以将另一个标准用于数据排列，如一个条目

时间戳，那么应该在查询时对数据进行排列，并且不存储显示顺序。如果显示顺序改变了呢？数据显示的顺序很可能不是数据的一个组成部分，而是一个正交问题。映射持久化列表之前，应该慎重；Hibernate 并非想象的那么智能，如下一个示例所示。

首先，修改 Item 实体及其集合属性。见代码清单 7.3。

代码清单7.3 持久化列表，在数据库中保存元素顺序
路径：/model/src/main/java/org/jpwh/model/collections/listofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @OrderColumn
    @Column(name = "FILENAME")
    protected List<String> images = new ArrayList<String>();
    // ...
}
```

启用持久化顺序：默认为
IMAGES_ORDER

←

这个示例中有一个新注解：`@OrderColumn`。该列会在持久化列表中存储索引，从零开始。注意，Hibernate 会在数据库中存储索引并且期望该索引是连续的。如果索引存在间隔，那么 Hibernate 将在加载和构造该列表时添加 null 元素。看看图 7-4 所示的架构。

ITEM	
ID	NAME
1	Foo
2	B
3	C

IMAGE		
ITEM_ID	IMAGES_ORDER	FILENAME
1	0	foo.jpg
1	1	bar.jpg
1	2	baz.jpg
1	3	baz.jpg
2	0	b1jpg
2	1	b2.jpg

图 7-4 保存每个列表元素位置的集合表

IMAGE 表的主键是 ITEM_ID 和 IMAGES_ORDER 的组合。这使重复 FILENAME 值成为可能，它是符合 List 的语意的。

之前说过，Hibernate 并非想象的那么智能。考虑修改该列表：假设该列表有 3 个元素 A、B 和 C，其顺序也是 A、B、C。如果从该列表中移除 A 会发生什么？Hibernate 会为该执行一个 SQL 语句 DELETE。然后它会为 B 和 C 执行两个 UPDATE，将它们的位置向左移动以便填补索引中的间隔。对于所删除元素右侧的每一个元素，Hibernate 都会执行 UPDATE 语句。如果为此手动编写 SQL，则可以用 UPDATE 语句完成该任务。在列表中间插入也是一样。Hibernate 会逐个将所有现有元素移动到右边。至少 Hibernate 足够智能，知道在 clear()列表时执行单个 DELETE。

现在，假定一个商品的图片除了文件名之外还有用户提供的名称。在 Java 中对此建模的一种方式是使用映射，利用键/值对。

7.1.7 映射一个映射

同样，对 Java 类做个小修改以便正确使用 Map 属性。见代码清单 7.4。

代码清单7.4 存储其键值对的持久化映射
路径: /model/src/main/java/org/jpwh/model/collections/mapofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    protected Map<String, String> images = new HashMap<String, String>();
    // ...
}
```

每个映射条目都是键/值对。这里要使用@MapKeyColumn 将键映射到 FILENAME❶并且将值映射到 IMAGENAME 列❷。这意味着用户仅可以使用文件一次，因为 Map 不允许重复键。

正如可以从图 7-5 的架构中所看到的，该集合表的主键是 ITEM_ID 和 FILENAME 的组合。该示例使用 String 作为该映射的主键；但 Hibernate 支持任意基本类型，如 BigDecimal 和 Integer。如果键是 Java 枚举 e num，则必须使用@MapKeyEnumerated。对于 java.util.Date 等所有时序类型，要使用@MapKey 则 Temporal。5.1.6 和 5.1.7 小节介绍过这些选项，只不过当时是用于集合。

ITEM	
ID	NAME
1	Foo
2	B
3	C

IMAGE		
ITEM_ID	FILENAME	IMAGENAME
1	foo.jpg	Foo
1	bar.jpg	Bar
1	baz.jpg	Baz
2	b1.jpg	B1
2	b2.jpg	B2

图 7-5 用于一个映射的表，使用字符串作为索引和元素

前一个示例中的映射是无序的。如果要根据文件名对映射条目进行排列，又该做什么呢？见代码清单 7.5。

Hibernate 特性

7.1.8 排列和排序集合

在英语令人惊讶的滥用中，排列(sorted)和排序(ordered)这两个词在用于 Hibernate 的持久化集合时意味着不同的东西。使用 Java 比较器在内存中对集合进行排列。在从数据库加载集合时，要使用带有 ORDER BY 子句的 SQL 查询对其排序。

将图片的映射变成排列过的映射。需要修改该 Java 属性和映射。见代码清单 7.5。

代码清单7.5 使用比较器在内存中对映射条目进行排列

路径: /model/src/main/java/org/jpwh/model/collections/sortedmapofstrings/Item.

java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.SortComparator(ReverseStringComparator.class)
    protected SortedMap<String, String> images =
        new TreeMap<String, String>();
    // ...
}
```

排列过的集合是一个 Hibernate 功能；因此用 `org.hibernate.annotations.SortComparator` 注解了 `java.util.Comparator` 的实现。这里不会介绍这个普通的类；它会以相反顺序对字符串进行排列。

数据库架构不会发生变化，所有后续示例的情况也是如此。如果需要回顾，可以看看前面几节中的介绍。

映射 `java.util.SortedSet` 如代码清单 7.6 所示。

代码清单7.6 使用String#compareTo()在内存中对集元素进行排列

路径: /model/src/main/java/org/jpwh/model/collections/sortedsetofstrings/Item.

java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @org.hibernate.annotations.SortNatural
    protected SortedSet<String> images = new TreeSet<String>();
    // ...
}
```

这里使用了自然排列，借助了 `String#compareTo()` 方法。

遗憾的是，无法对包进行排列；没有 `TreeBag`。列表元素的索引预定义了其顺序。

或者，相较于切换到 `Sorted*` 接口，可能希望以正确的顺序从数据库中检索集合的元素，而不在内存中排列。下列没有使用 `java.util.SortedSet`，而是使用了 `java.util.LinkedHashSet`。见代码清单 7.7。

代码清单 7.7 LinkedHashSet 为迭代提供了插入顺序

路径: /model/src/main/java/org/jpwh/model/collections/setofstringsorderby/Item.java

```

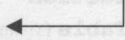
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    // @javax.persistence.OrderBy
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    protected Set<String> images = new LinkedHashSet<String>();

    // ...
}

```

只有一个可能的顺序:
FILENAME asc



LinkedHashSet 类有一个关于其元素的固定迭代顺序，而 Hibernate 会在加载集合时按正确的顺序填充它。为此，Hibernate 会将 ORDER BY 子句应用到加载该集合的 SQL 语句。必须使用专有的 `@org.hibernate.annotations.OrderBy` 注解声明这个 SQL 子句。甚至可以调用 SQL 函数，如 `@OrderBy("substring(FILENAME, 0, 3)desc")`，它会根据文件名的前三个字母进行排列。要小心检查 DBMS 是否支持调用的 SQL 函数。此外，可以使用 SQL:2003 语法 ORDER BY ... NULLS FIRST|LAST，并且 Hibernate 会自动将它转换成 DBMS 支持的方言。

Hibernate @OrderBy 与 JPA @OrderBy 的对比

可以将注解 `@org.hibernate.annotations.OrderBy` 应用到任何集合；它的参数是 Hibernate 附加到加载集合的 SQL 语句的一段普通 SQL。Java 持久化具有类似的注解：`@javax.persistence.OrderBy`。它(仅有)的参数不是 SQL 而是 `somePropertyDESC|ASC`。String 或 Integer 元素值没有属性。因此，当在基本类型集合上应用 JPA 的 `@OrderBy` 注解时，就像上一个示例中使用 `Set<String>` 一样，该规范规定，“将按照基本对象的值排序”。这意味着不能修改该顺序：在上一个示例中，其顺序在所生成的 SQL 查询中将总是按照 FILENAME asc 排序。在稍后的 7.2.2 小节中，将在元素值类具有持久化属性并且不是基本/标量类型时使用该 JPA 注解。

代码清单 7.8 所示的示例显示了加载包映射时的相同排序。

代码清单 7.8 ArrayList 提供了固定的迭代顺序

路径: /model/src/main/java/org/jpwh/model/collections/bagofstringsorderby/Item.java

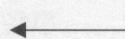
```

@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @org.hibernate.annotations.CollectionId(

```

代理主键允许重复



```
        columns = @Column(name = "IMAGE_ID"),
        type = @org.hibernate.annotations.Type(type = "long"),
        generator = Constants.ID_GENERATOR)
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    protected Collection<String> images = new ArrayList<String>();

    // ...
}
```

最后，可以使用 LinkedHashMap 加载排序后的键/值对。见代码清单 7.9。

代码清单7.9 LinkedHashMap按顺序保持键/值对

路径: /model/src/main/java/org/jpwh/model/collections/mapofstringsorderby/Item.
java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    protected Map<String, String> images = new LinkedHashMap<String,
        String>();

    // ...
}
```

要牢记，排序后集合的元素在被加载时仅会按照预期的顺序排序。只要添加并移除了元素，集合的迭代顺序就可能与“按照文件名的顺序”不同；它们的行为会像常规联结集、映射或列表。

在真实系统中，可能需要保留的对象比图片名称和文件名要多。很可能需要为额外信息创建 Image 类。这就是用于组件集合的完美用例。

7.2 组件集合

之前映射了一个可嵌入组件：第 5.2 节中 User 的 address(地址)。当前的情况有所不同，因为 Item 有许多对 Image 的引用，如图 7-6 所示。该 UML 图中的关联是一个组合(黑钻石，意即最标准的形式)；因此，所引用的 Images 会被绑定到所属 Item 的生命周期。

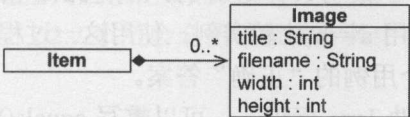


图 7-6 Item 中 Image 组件的集合

代码清单 7.10 中的代码显示了新的 Image 可嵌入类，以便捕获所关注图片的所有属性。

代码清单7.10 封装图片的所有属性

路径: /model/src/main/java/org/jpwh/model/collections/setofembeddables/Image.
java

```
@Embeddable
public class Image {

    @Column(nullable = false)
    protected String title;

    @Column(nullable = false)
    protected String filename;

    protected int width;

    protected int height;
    // ...
}
```

首先, 注意所有的属性都是非可选的, NOT NULL。尺寸属性不可为空, 因为它们的值是基元。其次, 必须考虑相等性以及数据库和 Java 层如何比较两张图片。

7.2.1 组件实例的相等性

假设要在 HashSet 中保存几个 Image 实例。知道集不允许重复元素, 那么集如何检测重复呢? HashSet 会调用放置在 Set 中每个 Image 上的 equals()方法(显然, 它还会调用 hashCode()方法来获取)。以下集合中有多少图片?

```
someItem.getImages().add(new Image(
    "Foo", "foo.jpg", 640, 480
));
someItem.getImages().add(new Image(
    "Bar", "bar.jpg", 800, 600
));
someItem.getImages().add(new Image(
    "Baz", "baz.jpg", 1024, 768
));
someItem.getImages().add(new Image(
    "Baz", "baz.jpg", 1024, 768
));
assertEquals(someItem.getImages().size(), 3);
```

期望得到 4 张图片而非 3 张吗? 你是对的: 常规 Java 相等性检查依赖于标识。java.lang.Object#equals()方法会使用 $a=b$ 比较实例。使用这一过程, 集合中就会有 Image 的 4 个实例。显然, 3 张才是这个用例的“正确”答案。

对于 Image 类, 不必借助 Java 标识——可以重写 equals()和 hashCode()方法。见代码清单 7.11。

代码清单7.11 使用equals()和hashCode()实现自定义相等性

路径: /model/src/main/java/org/jpwh/model/collections/setofembeddables/Image.

java

@Embeddable

public class Image {

@Override

```
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
```

①相等性检查

```
    Image other = (Image) o;
```

```
    if (!title.equals(other.title)) return false;
    if (!filename.equals(other.filename)) return false;
    if (width != other.width) return false;
    if (height != other.height) return false;
```

```
    return true;
```

}

@Override

```
public int hashCode() {
    int result = title.hashCode();
    result = 31 * result + filename.hashCode();
    result = 31 * result + width;
    result = 31 * result + height;
    return result;
```

②必须是对称的

}

// ...

这个在 equals()①中的自定义相等性检查会将一个 Image 的所有值和另一个 Image 的值进行比较。如果所有的值都相同，那么图片就必定是相同的。hashCode()②方法必须是对称的；如果两个实例相等，则它们必定具有相同的哈希代码。

为何在 5.2 节中没有在映射 User 的 Address 之前重写相等性呢？好吧，真相是大概应该那样做。唯一的理由是，使用常规标识相等性不会有任何问题，除非将可嵌入组件放入 Set 或者在 Map 中将其用作键。然后应该根据值而非标识来重新定义相等性。最好是重写每个 @Embeddable 类上的这些方法；应该“根据值”来比较所有的值类型。

现在思考一下数据库主键：Hibernate 会生成一个将 IMAGE 集合表的所有不可为空列包含在组合主键中的架构。这些列必须是不可为空的，因为不能标识不知道的对象。这反映了 Java 类中的相等性实现。7.2.2 小节会介绍该架构，其中包含更多与主键有关的细节。

提示：

我们必须提及使用 Hibernate 架构生成器的小问题：如果使用 @NotNull 而非 @Column(nullable=false) 注解可嵌入的属性，Hibernate 将不会为集合表的列生成 NOT NULL 约束。实例的 Bean 验证(Bean Validation)检查会按预期运行，只有数据库架构缺少

了完整性规则。如果可嵌入类是在集合中映射的，则要使用 `@Column(nullable=false)`，并且属性应该是主键的一部分。

现在该组件类就准备好了，可以在集合映射中使用它。

7.2.2 组件集

映射组件的集，如代码清单 7.12 所示。

代码清单7.12 重写的可嵌入组件集

路径: /model/src/main/java/org/jpwh/model/collections/setofembeddables/Item.java

```
@Entity
public class Item {

    @ElementCollection                                ← ①必须的
    @CollectionTable(name = "IMAGE") ← ②重写集合表名称
    @AttributeOverride(
        name = "filename",
        column = @Column(name = "FNAME", nullable = false)
    )
    protected Set<Image> images = new HashSet<Image>();

    // ...
}
```

像之前一样，`@ElementCollection` 注解①是必须的。Hibernate 会从泛型集合声明中自动获悉该集合的目标是 `@Embeddable` 类型。`@CollectionTable` 注解②会重写该集合表的默认名称，该名称将会是 `ITEM_IMAGES`。

`Image` 映射会定义集合表的列。就像对于单个嵌入值所做的那样，可以使用 `@AttributeOverride` 自定义映射，而无须修改目标可嵌入类。看看图 7-7 所示的数据库架构。

ITEM	
ID	NAME
1	Foo
2	B
3	C

IMAGE				
ITEM_ID	TITLE	FNAME	WIDTH	HEIGHT
1	Foo	foo.jpg	640	480
1	Bar	bar.jpg	800	600
1	Baz	baz.jpg	1024	768
2	B	b.jpg	640	480

图 7-7 用于一个组件集合的示例数据表

正在映射一个集，所以该集合表的主键是外键列 `ITEM_ID` 和所有“嵌入的”不可为空列：`TITLE`、`FNAME`、`WIDTH` 和 `HEIGHT`。

`ITEM_ID` 值并未包含在重写的 `Image` 的 `equals()` 和 `hashCode()` 方法中，正如 7.2.1 小节中所探讨过的。因此，如果在集中混合不同商品的图片，那么将在 Java 层中遇到相等性的问题。在该数据库表中，显然可以区分不同商品的图片，因为主键相等性检查中包含了商品的标识符。

如果希望在 Image 的相等性例程中包含该 Item，变成与数据库主键对称，则需要一个 Image#item 属性。这是一个简单的后向指针，它由 Hibernate 在加载 Image 实例时提供：

路径：/model/src/main/java/org/jpwh/model/collections/setofembeddables/
Image.java

```
@Embeddable
public class Image {

    @org.hibernate.annotations.Parent
    protected Item item;
    // ...
}
```

现在可以在 equals() 和 hashCode() 实现中获取父 Item 值，并编写一个比较，例如使用 this.getItem().getId().equals(other.getItem().getId()) 编写。要当心 Item 未被持久化且没有标识符值的情况；将在第 10.3.2 节中更深入探究这个问题。

如果需要元素的加载时排序以及使用 LinkedHashSet 的稳定迭代顺序，则可以使用 JPA 的 @OrderBy 注解：

路径：/model/src/main/java/org/jpwh/model/collections/setofembeddablesorderby/
Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @OrderBy("filename, width DESC")
    protected Set<Image> images = new LinkedHashSet<Image>();
    // ...
}
```

@OrderBy 注解的参数是 Image 类的属性，后面跟着表示升序的 ASC 或者表示降序的 DESC。默认是升序，因此这个示例会根据图片文件名升序排列，然后根据每张图片的宽度降序排列。注意这与专有的 @org.hibernate.annotations.OrderBy 注解不同，该注解会使用普通 SQL 子句，7.1.8 小节将进行探讨。

将 Image 的所有属性声明为 @NotNull 可能并非你所希望的。如果任何属性都是可选的，就需要用于集合表的不同主键。

7.2.3 组件包

之前使用了 @org.hibernate.annotations.CollectionId 注解将代理键列添加到集合表。不过，该集合类型并非 Set 而是 Collection——一个包。这与更新后的架构是一致的：如果有代理主键列，那么重复“元素值”是允许的。下面用一个示例来阐释这一点。

首先，Image 类现在可以使用可为空的属性：

路径: /model/src/main/java/org/jpwh/model/collections/bagofembeddables/Image.
java

```
@Embeddable
public class Image {

    @Column(nullable = true)
    protected String title;

    @Column(nullable = false)
    protected String filename;

    protected int width;

    protected int height;

    // ...
}
```

如果有一个代理主键，则可以为 null

记住，当“根据值”对比实例时，要在重写的 equals()和 hashCode()方法中说明 Image 的可选 title。

接下来，看看 Item 中包集合的映射：

路径: /model/src/main/java/org/jpwh/model/collections/bagofembeddables/Item.
java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @org.hibernate.annotations.CollectionId(
        columns = @Column(name = "IMAGE_ID"),
        type = @org.hibernate.annotations.Type(type = "long"),
        generator = Constants.ID_GENERATOR)
    protected Collection<Image> images = new ArrayList<Image>();

    // ...
}
```

Hibernate 特性

就像之前在 7.1.5 小节中一样，要使用专有的@org.hibernate.annotations.CollectionId 注解声明额外的代理主键列 IMAGE_ID。图 7-8 显示了该数据库架构。

ITEM	
ID	NAME
1	Foo
2	B
3	C

IMAGE					
IMAGE_ID	ITEM_ID	TITLE	FILENAME	WIDTH	HEIGHT
1	1	Foo	foo.jpg	640	480
2	1		bar.jpg	800	600
3	1	Baz	baz.jpg	1024	768
4	1	Baz	baz.jpg	1024	768
5	2	B	b.jpg	640	480

图 7-8 具有代理主键列的组件表集合

标识符为 2 的 Image 的 title 为 null。
接下来，看看变更带有 Map 的集合表主键的另一种方式。

7.2.4 组件值的映射

如果 Images 存储在 Map 中，则文件名可以是映射键：

路径：/model/src/main/java/org/jpwh/model/collections/mapofstringembeddables/
Item.java

```
@Entity
public class Item {

    @ElementCollection                可选，默认为 IMAGES_KEY
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    protected Map<String, Image> images = new HashMap<String, Image>();

    // ...
}
```

如图 7-9 所示，现在该集合表的主键是外键列 ITEM_ID 及映射的键列 FILENAME。

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	FILENAME	TITLE	WIDTH	HEIGHT
1	foo.jpg	Foo	640	480
1	bar.jpg		800	600
1	baz.jpg	Baz	1024	768
2	b.jpg	B	640	480

图 7-9 用于组件映射的数据库表

可嵌入 Image 类会映射所有其他的列，它可以为空：

路径：/model/src/main/java/org/jpwh/model/collections/mapofstringembeddables/
Image.java

```
@Embeddable
public class Image {

    @Column(nullable = true)
    protected String title;
    protected int width;
    protected int height;

    // ...
}
```

在上一个示例中，映射中的值是可嵌入组件类的实例，并且映射的键是一个基本字符串。接下来，要将可嵌入类型同时用于键和值。

7.2.5 作为映射键的组件

最后一个示例是同时使用可嵌入类型的键和值映射 Map，如图 7-10 所示。

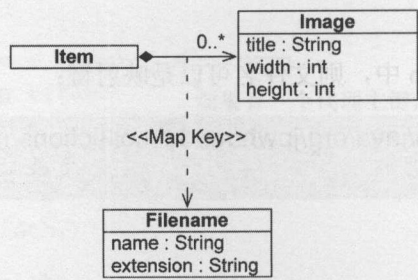


图 7-10 Item 有以 Filename 为键的 Map

这里不使用字符串表示形式，可以使用自定义类型来表示文件名，如代码清单 7.13 所示。

代码清单7.13 用自定义类型表示文件名

路径: /model/src/main/java/org/jpwh/model/collections/mapofembeddables/ Filename.
java

@Embeddable

public class Filename {

 @Column(nullable = false)

 protected String name;

 @Column(nullable = false)

 protected String extension;

 @Override

 public boolean equals(Object o) {

 if (this == o) return true;

 if (o == null || getClass() != o.getClass()) return false;

 Filename filename = (Filename) o;

 if (!extension.equals(filename.extension)) return false;

 if (!name.equals(filename.name)) return false;

 return true;

 }

 @Override

 public int hashCode() {

 int result = name.hashCode();

 result = 31 * result + extension.hashCode();

 return result;

 }

 // ...

}

← 必须是 NOT NULL:
← 主键的一部分

如果希望将这个类用作一个映射的键，那么所映射的数据库列就不能是可为空，因为它们都是组合主键的一部分。还必须重写 equals()和 hashCode()方法，因为映射的键是一个集，并且指定键集中的每个 Filename 都必须是唯一的。

无须任何特殊注解来映射该集合：

路径：/model/src/main/java/org/jpwh/model/collections/mapofembeddables/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    protected Map<Filename, Image> images = new HashMap<Filename, Image>();

    // ...
}
```

实际上，不能应用@MapKeyColumn 和@AttributeOverrides；当映射的键是@Embeddable 类时，它们没有任何效果。IMAGE 表的组合主键包括 ITEM_ID、NAME 和 EXTENSION 列，正如可以在图 7-11 中看到的。

ITEM	
ID	NAME
1	Foo
2	B
3	C

IMAGE					
ITEM_ID	NAME	EXTENSION	TITLE	WIDTH	HEIGHT
1	foo	jpg	Foo	640	480
1	bar	jpg		800	600
1	baz	jpg	Baz	1024	768
2	b	jpg	B	640	480

图 7-11 用于以 Filenames 为键的 Images 的 Map 的数据库表

像 Image 这样的组合可嵌入类没有被限制为基本类型的简单属性。已经看到过如何嵌套其他组件，如 Address 中的 City。可以提取和封装新的 Dimensions 类中 Image 的宽度和高度属性。

可嵌入类还可以拥有集合。

7.2.6 可嵌入组件中的集合

假定对于每个 Address，都希望存储一个联系人的列表。这在该可嵌入类中是一个简单 Set<String>：

路径：/model/src/main/java/org/jpwh/model/collections/embeddablesetofstrings/Address.java

```
@Embeddable
public class Address {

    @NotNull
    @Column(nullable = false)
```

```
protected String street;

@NotNull
@Column(nullable = false, length = 5)
protected String zipcode;

@NotNull
@Column(nullable = false)
protected String city;

@ElementCollection
@CollectionTable(
    name = "CONTACT",
    joinColumns = @JoinColumn(name = "USER_ID"))

@Column(name = "NAME", nullable = false)
protected Set<String> contacts = new HashSet<String>();
// ...
}
```

默认为 USER_CONTACTS

默认

默认为 CONTACTS

@ElementCollection 是唯一需要的注解；表和列名称都有默认值。看看图 7-12 中的架构：USER_ID 列有一个引用所属实体表 USERS 的外键约束。该集合表的主键是 USER_ID 和 NAME 列的组合，避免出现适合 Set 的重复元素。

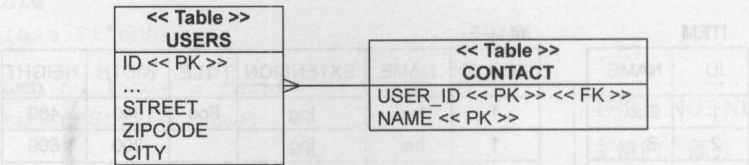


图 7-12 USER_ID 有一个引用 USERS 的外键约束

不用 Set，而是映射一个列表、包或者基本类型的映射。Hibernate 还支持可嵌入类型的集合，所以不用简单联系人字符串，而是编写可嵌入的 Contact 类并且让 Address 持有 Contacts 的集合。

尽管 Hibernate 认人非常灵活地使用组件映射和细粒度模型，但要意识到代码的阅读次数通常比编写次数要多。想想几年后接手必须维护这些代码的开发人员吧。

转换一下关注点，现在将注意力放在实体关联上：特别是简单多对一和一对多关联。

7.3 映射实体关联

在本章开头，承诺过要探讨父/子关系。到目前为止，已经映射了一个实体——Item。假设这是父。它有一个子集合：Image 实例的集合。父/子这个术语意味着某种生命周期依赖，因此字符串或可嵌入组件的集合是合适的。子是完全依赖于父的；它们将总是随着父一起而绝不会单独被保存、更新和移除。现在已经映射了一个父/子关系！父是一个实体，而许多子是值类型。

现在希望映射另一种关系：两个实体类之间的关联。它们的实例没有依赖性的生命周期。实例可以被保存、更新和移除，而不会影响其他任何实例。自然，有时候即便是实体

实例之间也有依赖关系。需要对两个类之间的关系如何影响实例状态有更为细粒度的控制，而不是完全取决于(嵌入的)类型。我们仍然在探讨父/子关系吗？事实证明父/子是模糊不清的，每个人都有其自己的定义。从现在开始，将尝试不使用术语，而是借助更精确或至少明确定义的词汇。

在后面几节中要探究的关系总是相同的，介于 Item 和 Bid 实体类之间，如图 7-13 所示。从 Bid 到 Item 的关联是多对一关联。稍后将让这一关联变成双向的，因此从 Item 到 Bid 的反向关联是一对多的。

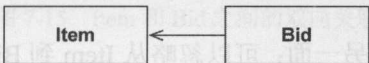


图 7-13 Item 和 Bid 之间的关系

多对一关联是最简单的，所以首先讨论它。其他的关联——多对多和一对一——更为复杂，将在第 8 章中探讨。

先来看看多对一关联。

7.3.1 最简单的可能关联

将 Bid#item 属性的映射称为单向的多对一关联。在探讨这一映射之前，先看看图 7-14 所示的数据库架构。

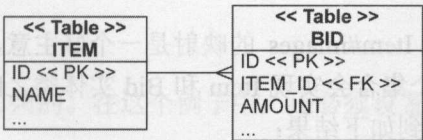


图 7-14 SQL 架构中的多对一关系

代码清单7.14 Bid具有单个对Item的引用

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/bidirectional/
Bid.java

```
@Entity
public class Bid {

    @ManyToOne(fetch = FetchType.LAZY) ← 默认为 EAGER
    @JoinColumn(name = "ITEM_ID", nullable = false)
    protected Item item;

    // ...
}
```

@ManyToOne 注解会将属性标记为实体关联，并且它是必须的。遗憾的是，其 fetch 参数会默认为 EAGER：这意味着无论何时加载 Bid 都会加载关联的 Item。我们通常偏好将延迟加载作为默认策略，稍后将在 12.1.1 节中探讨更多相关内容。

多对一实体关联会自然而然地映射到外键列：BID 表中的 ITEM_ID。在 JPA 中，这被称为联结列。该属性上不需要其他注解，只要@ManyToOne 注解即可。该联结列的默认名

称是 `ITEM_ID`: Hibernate 会自动使用目标实体名称及其标识符属性的组合, 用下划线分隔。

可以使用 `@JoinColumn` 注解重写该外键列。这里使用它有另一个原因: 在 Hibernate 生成 SQL 架构时让该外键列变成 `NOT NULL`。出价必须始终有对商品的引用; 它无法独立存在(注意, 这已经表明了必须牢记的某种生命周期依赖)。或者, 可以使用 `@ManyToOne(optional = false)` 或者像平常一样使用 Bean 验证的 `@NotNull`, 将关联标记为非可选。

这很容易。认识到可以编写一个完整且复杂的应用程序而无须使用其他任何方法是至关重要的。

你不需要映射这一关系的另一面; 可以忽略从 `Item` 到 `Bid` 的一对多关联。该数据库架构中只有一个外键列, 并且已经映射过它了。对此要认真看待: 当看见涉及的外键列和两个实体类时, 可能应该使用 `@ManyToOne` 而非其他注解来映射它。现在可以通过调用 `someBid.getItem()` 得到每个 `Bid` 的 `Item`。JPA 提供程序将解除该外键的引用并且加载 `Item`; 它还会负责管理外键值。如何得到一个商品的所有出价呢? 可以用 Hibernate 所支持的任何一种查询语言编写查询并且使用 `EntityManager` 执行。例如, 在 JPQL 中, 可以使用 `select b from Bid b where b.item = :itemParameter`。当然, 使用像 Hibernate 这样的完整 ORM 工具的一个原因是, 不希望亲自编写和执行查询。

7.3.2 让其变成双向的

在本章开头, 对于集合 `Item#images` 的映射是一个好主意, 给出了很多理由。对于集合 `Item#bids` 也是如此。这个集合会实现 `Item` 和 `Bid` 实体类之间的一对多关联。如果创建并映射这一集合属性, 将得到如下结果:

- Hibernate 会在调用 `someItem.getBids()` 并且开始遍历集合元素时自动执行 SQL 查询 `SELECT * from BID where ITEM_ID = ?`。
- 可以在集合中串联从一个 `Item` 到所有引用的 `Bids` 的状态变化。可以选择哪些生命周期事件应该被传递: 例如, 可以声明, 在保存 `Item` 时, 所有引用的 `Bid` 实例都应该被保存, 这样就不必为所有出价重复调用 `EntityManager#persist()` 了。

这并非一份很长的代码清单。一对多映射的主要好处是对数据的可导航访问。它是 ORM 的一个核心保障, 让你仅通过调用 Java 域模型的方法就能够访问数据。当处理你自己设计的高级别接口 `someItem.getBids().iterator().next().getAmount()` 等时, ORM 引擎应该负责以巧妙的方式加载所需要的数据。

能有选择地串联对相关实例的一些状态变更是不错的意外收获。不过, 要考虑到一些依赖性表明了 Java 级别的值类型, 而非实体。问问你自己, 架构中的所有表是否都有 `BID_ID` 外键列。如果不是, 则将 `Bid` 类映射为 `@Embeddable`, 而非 `@Entity`, 像之前那样使用一些表, 但要使用具有用于可传递状态变更的混合规则的不同映射。如果其他任何表在任意 `BID` 行上有外键引用, 就需要共享的 `Bid` 实体; 它不能被映射为嵌入 `Item`。

那么, 是否应该映射 `Item#bids` 集合呢? 你会得到可导航的数据访问, 但所付出的代价就是额外的 Java 代码以及更显著的复杂性。这常常是一个很困难的抉择。在应用程序中会有多频繁地调用 `someItem.getBids()` 然后按照预定义顺序访问/显示所有出价? 如果仅仅

希望显示出价的一个子集，或者如果每次都需要按不同顺序检索它们，那么无论如何都需要手动编写和执行查询。一对多映射及其集合将仅仅成为维护的包袱。根据经验，这是问题和缺陷的常见来源，尤其对于 ORM 初学者来说，更是如此。

在 CaveatEmptor 的例子中，答案是肯定的，会频繁调用 `someItem.getBids()`，然后将列表显示给希望参与拍卖的用户。图 7-15 显示了具有这一双向关联的更新后的 UML 图。

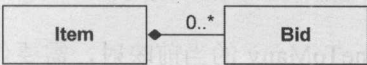


图 7-15 Item 和 Bid 之间的双向关联

该集合的映射和一对多方面的代码如代码清单 7.15 所示。

代码清单 7.15 Item 具有 Bid 引用的集合

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/ bidirectional/Item.
java

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item",
               fetch = FetchType.LAZY)
    protected Set<Bid> bids = new HashSet<>();
    // ...
}
```

双向关联所必须的

默认

`@OneToMany` 注解是必须的。在这个例子中，还必须设置 `mappedBy` 参数。该参数是“另一侧”上属性的名称。

再看看另一侧：代码清单 7.15 中的多对一映射。Bid 类中的属性名称是 `item`。bid 侧承担了外键列 `ITEM_ID` 的职责，要使用 `@ManyToOne` 来映射它。`mappedBy` 会告知 Hibernate “使用已经由给定属性映射的外键列来加载这个集合”——在这个例子中是 `Bid#item`。在一对多是双向关联且已经映射了该外键列时，`mappedBy` 参数总是必须的。将在第 8 章中再次探讨它。

用于集合映射的 `fetch` 参数的默认设置始终是 `FetchType.LAZY`。未来不需要此选项。它是很好的默认设置；其相反的选项是很少用到的 `EAGER`。你不会希望每次加载 Item 时都急加载所有的出价。应该在访问时按需要加载它们。

映射 `Item#bids` 集合的第二个原因是串联状态变更的能力。

7.3.3 级联状态

如果可以跨越到另一个实体的关联来级联实体的状态变更，就只需要较少的代码行来管理关系。以下代码会创建一个新的 Item 和一个新的 Bid，然后链接它们：

```
Item someItem = new Item("Some Item");

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);
```

不要忘记！

必须考虑这一关系的两面：`Bid` 构造函数接受一个商品，用于填充 `Bid#item`。要维护内存中实例的完整性，需要将出价添加到 `Item#bids`。现在从 Java 代码的角度来看，该链接已经完成了；所有的引用都设置好了。如果不确定为何需要这段代码，参阅 3.2.4 小节。

在数据库中保存该商品及其出价，先不使用传递持久化，以后再使用它。

1. 启用传递持久化

使用 `@ManyToOne` 和 `@OneToMany` 的当前映射，需要代码清单 7.16 所示的代码来保存一个新的 `Item` 和几个 `Bid` 实例。

代码清单 7.16 分别管理独立的实体实例

路径: `/examples/src/test/java/org/jpwh/test/associations/OneToManyBidirectional.java`

```
Item someItem = new Item("Some Item");
em.persist(someItem);

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);           ← 不要忘记!
em.persist(someBid);

Bid secondBid = new Bid(new BigDecimal("456.00"), someItem);
someItem.getBids().add(secondBid);
em.persist(secondBid);

tx.commit();                               ← 脏检查: SQL 执行
```

创建几个出价时，在每个出价上调用 `persist()` 似乎有些多余。新的实例是临时的并且必须被持久化。`Bid` 和 `Item` 之间的关系不会影响它们的生命周期。如果 `Bid` 成为值类型，则 `Bid` 的状态就会自动变成与其所属 `Item` 一样。不过，在这个例子中，`Bid` 有其自己完全独立的状态。

之前介绍过，细粒度控制有时对于表达关联实体类之间的依赖性是不够的；该例就是。JPA 中用于此的机制是 `cascade` 选项。例如，要在保存商品时保存所有的出价，则要映射该集合，代码如代码清单 7.17 所示。

代码清单 7.17 级联从 `Item` 到所有出价的持久化状态

路径: `/model/src/main/java/org/jpwh/model/associations/onetomany/cascadepersist/Item.java`

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item", cascade = CascadeType.PERSIST)
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```


级联选项要用于想要传递的每个操作，因此要将 `CascadeType.PERSIST` 用于 `EntityManager#persist()`操作。现在可以简化链接商品和出价以及之后保存它们的代码。见代码清单 7.18。

代码清单7.18 所有引用的bids都会自动持久化
路径: /examples/src/test/java/org/jpwh/test/associations/OneToManyCascadePersist.java

```
Item someItem = new Item("Some Item");
em.persist(someItem);

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);

Bid secondBid = new Bid(new BigDecimal("456.00"), someItem);
someItem.getBids().add(secondBid);

tx.commit();
```

自动保存出价(稍后, 在刷新时)

脏检查; SQL 执行

在提交时, Hibernate 会检查管理/持久化的 Item 实例并细查 bids 集合。之后它会从内部在每个引用的 Bid 实例上调用 `persist()`并且保存它们。BID#ITEM_ID 列中存储的值会通过检查 Bid#item 属性从每个 Bid 中取出。外键列是通过在该属性上使用 `@ManyToOne` 来“映射”的。

`@ManyToOne` 注解也有 `cascade` 选项。通常不会使用它。例如, 真的不能说“在保存出价时, 也保存商品”。该商品必须先存在; 否则, 该出价在数据库中就是无效的。思考一下另一个可能的 `@ManyToOne`: `Item#seller` 属性。User 必须在能出售 Item 前就存在。

传递持久化是一个简单概念, 对于使用 `@OneToMany` 或 `@ManyToMany` 映射通常很有用。另一方面, 必须谨慎应用传递删除。

2. 级联删除

删除商品就表明删除该商品的所有出价, 这看起来是合理的, 因为它们独立存在时不再具有相关性。这就是组合(填充钻石, 意即所填充的标准形式)在 UML 图中的意义。使用当前级联选项, 就必须编写以下代码来删除一个商品:

路径: /examples/src/test/java/org/jpwh/test/associations/OneToManyCascadePersist.java

```
Item item = em.find(Item.class, ITEM_ID);
for (Bid bid : item.getBids()) {
    em.remove(bid);
}
em.remove(item);
```

① 移除出价

② 移除所有者

首先移除出价①, 然后移除所有者: Item②。删除顺序很重要。如果首先移除 Item,

则会违反外键约束，因为 SQL 操作会按照 `remove()` 调用顺序排队。首先必须删除 BID 表中的行，然后删除 ITEM 表中的行。

JPA 提供了一个级联选项来帮助完成此任务。持久化引擎可以自动移除关联的实体实例。见代码清单 7.19。

代码清单 7.19 级联从 Item 到所有出价的移除

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/cascaderemove/Item.
java

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item",
        cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

就像之前使用 PERSIST 一样，Hibernate 现在会级联这个关联上的 `remove()` 操作。如果调用 Item 上的 `EntityManager#remove()`，Hibernate 会加载 bids 集合元素并且在每个实例上内部调用 `remove()`：

路径: /examples/src/test/java/org/jpwh/test/associations/OneToManyCascadeRemove.
java

```
Item item = em.find(Item.class, ITEM_ID);
em.remove(item);
```

在加载出价后逐个
删除它们

必须加载该集合，因为每个 Bid 都是独立的实体实例，并且必须经历常规的生命周期。如果 Bid 类上提供了 `@PreRemove` 回调方法，则 Hibernate 必须执行它。第 13 章将介绍更多与对象状态和回调有关的内容。

这一删除过程是低效的：Hibernate 必须总是加载集合并且分别删除每个 Bid。单个 SQL 语句在数据库上会有相同效果：`delete from BID where ITEM_ID = ?`。

明白这一点，因为数据库中没有对 BID 表的外键引用。Hibernate 不知道这一点，并且不能为可能有 BID_ID 的某个行搜索整个数据库。

如果选用了可嵌入组件的集合而非 `Item#bids`，那么 `someItem.getBids().clear()` 就会执行单个 SQL 语句 DELETE。有了值类型的集合，Hibernate 就会假定没什么东西可能持有对出价的引用，并且仅从集合中移除引用会让出价变成孤立的、可移除数据。

3. 启用孤立的移除

JPA 提供了一个(存在疑问的)标记，它可以为 `@OneToMany` (并且仅能为 `@OneToMany`) 实体关联启用相同的行为。见代码清单 7.20。

代码清单7.20 在@OneToMany集合上启用孤立的移除

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/orphanremoval/Item.
java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",
                cascade = CascadeType.PERSIST,      包括 CascadeType.REMOVE
                orphanRemoval = true)
    protected Set<Bid> bids = new HashSet<>();
    // ...
}
```

orphanRemoval=true 参数告知 Hibernate 希望在从集合中移除 Bid 时永久移除它。下面是删除单个 Bid 的示例:

路径: /examples/src/test/java/org/jpwh/test/associations/OneToManyOrphanRemoval.
java

```
Item item = em.find(Item.class, ITEM_ID);
Bid firstBid = item.getBids().iterator().next();
item.getBids().remove(firstBid);  ← 移除了一个出价
```

Hibernate 会监控该集合，并且事务中的提交将会注意到你从该集合中移除了一个元素。现在 Hibernate 会认为该 Bid 是孤立的。要确保没有其他东西引用它；唯一的引用就是刚刚从集合中移除的那个。Hibernate 会自动执行 SQL 语句 DELETE 移除数据库中的 Bid 实例。

仍旧无法像组件集合那样得到 clear()一次性 DELETE。Hibernate 注重常规的实体状态转换，且出价是全部加载以及单独移除的。

为何说孤立的移除存在疑问呢？当然，它在这个示例中是可用的。到目前为止数据库中没有任何表有 BID 的外键引用。从 BID 表中删除一行不会产生意外影响；唯一对出价的内存中引用位于 Item#bids 中。只要这些条件都满足，那么启用孤立的移除就没什么问题。例如，当表示层可以从一个集合中移除一个元素以便删除某些东西时，它就是一个便利的选项；你仅使用了域模型实例，并且你无须调用服务来执行这一操作。

思考一下，创建一个 User#bids 集合映射——另一个@OneToMany 时，会发生什么，如图 7-16 所示。这是验证你的 Hibernate 知识的好时机：在这一变更之后，表和架构看起来会是什么样子？(答案：BID 表会有一个引用 USERS 的 BIDDER_ID 外键列)。



图 7-16 Item、Bid 和 User 之间的双向关联

代码清单 7.21 中所示的测试将无法通过。

代码清单 7.21 Hibernate 不会在数据库移除之后清除内存中的引用

路径: /examples/src/test/java/org/jpwh/test/associations/OneToManyOrphanRemoval.
java

```
User user = em.find(User.class, USER_ID);
assertEquals(user.getBids().size(), 2);    ← 出价两次的用户

Item item = em.find(Item.class, ITEM_ID);
Bid firstBid = item.getBids().iterator().next();
item.getBids().remove(firstBid);          ← 移除了一个出价

// FAILURE!
// assertEquals(user.getBids().size(), 1);
assertEquals(user.getBids().size(), 2);    ← 还是两个!
```

Hibernate 会认为移除的 Bid 是孤立并且可删除的；在数据库中它将被自动删除，但仍旧在另一个集合 User#bids 中持有一个对它的引用。这一事务提交时，数据库状态是良好的；所删除的 BID 表的行包含两个外键：ITEM_ID 和 BIDDER_ID。内存中存在不一致性，因为“在从集合中移除引用的时候移除实体实例”这句话自然与共享引用有冲突。

相较于孤立的移除，甚至 CascadeType.REMOVE，应该总是考虑使用更简单的映射。此处，Item#bids 像使用 @ElementCollection 映射的组件集合一样好用。Bid 将是 @Embeddable 并且具有一个 @ManyToMany 属性，引用一个 User(可嵌入组件可以拥有对实体的单向关联)。

这会提供所寻求的生命周期：对所属实体的完全依赖。必须避免共享引用；UML 图(如图 7-16 所示)会让从 Bid 到 User 的关联变成单向的。删除 User#bids 集合；不需要这个 @OneToMany。如果需要用户做出的所有出价，则可以编写一个查询：select b from Bid b where b.bidder = :userParameter(在下一章中，你要在一个可嵌入组件中使用 @ManyToOne 完成这个映射)。

Hibernate 特性

4. 在外键上启用 ON DELETE CASCADE

介绍过的所有移除操作都是低效的；必须在内存中加载出价，并且需要许多 SQL 的 DELETE 语句。SQL 数据库支持更加高效的外键功能：ON DELETE 选项。在 DDL 中，它看起来是这样的：用于 BID 表的 foreign key(ITEM_ID) references ITEM on delete cascade。

这个选项告知数据库为所有访问数据库的应用程序透明地保持组合的引用完整性。无论何时删除 ITEM 表中的行，数据库都会自动删除 BID 表中具有相同 ITEM_ID 键值的所有行。只需要一个 DELETE 语句来递归移除所有的依赖数据，并且不需要在应用程序(服务器)内存中加载任何东西。

应该检查架构是否已经在外键上启用了这个选项。如果想要将这个选项添加到 Hibernate 生成的架构中，则可以使用 Hibernate@OnDelete 注解。见代码清单 7.22。

代码清单7.22 在架构中生成外键ON DELETE CASCADE

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/ondeletescascade/Item.
java

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item", cascade = CascadeType.PERSIST)

    @org.hibernate.annotations.OnDelete(
        action = org.hibernate.annotations.OnDeleteAction.CASCADE
    )
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

←
Hibernate 技巧: 模式选项
通常位于 mappedBy 侧。

这里可以看到 Hibernate 的一个技巧: `@OnDelete` 注解仅会影响 Hibernate 的架构生成。影响架构生成的设置通常位于“另一个”`mappedBy` 侧, 其中映射了外键/联结列。在 `Bid` 中, `@OnDelete` 注解通常位于 `@ManyToOne` 旁边。不过, 当关联被双向映射时, Hibernate 将仅识别位于 `@OneToMany` 侧的 `@OnDelete`。

在数据库中启用外键串联删除不会影响 Hibernate 的运行时行为。你仍会遇到代码清单 7.21 中所示的相同问题。内存中的数据可能不再准确反映数据库中的状态。当删除 ITEM 表中的一行时, 如果 BID 表中的所有相关行都被自动移除, 那么应用程序代码就要负责清除引用并且保持与数据库状态的一致。如果不仔细, 最终甚至可能会保存你或其他人之前删除过的内容。

`Bid` 实例不会经历常规的生命周期, 而且 `@PreRemove` 这样的回调没有任何效果。此外, Hibernate 不会自动清除可选的二级全局缓存, 它可能包含失效的数据。基本上, 使用数据库级别外键级联你可能遇到的各种问题, 会与除你的应用程序之外的另一个应用程序在访问相同数据库或其他数据库触发器引发变更时所遇到的问题相同。在这样的场景中, Hibernate 会是非常有效的实用工具, 但也有其他运行部件可以考虑。本书后续会探讨更多与并发和缓存有关的内容。

如果在处理一个新架构, 最容易的方法就是不启用数据库级别级联并且将域模型中的组合关系映射为嵌入的/可嵌入, 而不是映射为实体关联。然后 Hibernate 会执行有效的 SQL DELETE 操作来移除整个组合。7.3.2 小节建议过: 如果能避免共享引用, 则可以将 `Bid` 映射为 `Item` 中的 `@ElementCollection`, 而不是映射为有 `@ManyToOne` 和 `@OneToMany` 关联的独立实体。或者, 可能完全不映射任何集合, 并且仅适用最简单的映射: 有 `@ManyToOne` 的外键列, 在 `@Entity` 类之间是单向的。

7.4 本章小结

- 使用简单集合映射, 如 `Set<String>`, 期间使用了一组丰富的接口和实现。

高级实体关联映射

第 8 章

8

本章内容简介：

- 映射一对一实体关联
- 一对多映射选项
- 多对多和三元实体关系

第 7 章阐释了单向多对一关联，让其变成双向，并最终使用串联选项启用了传递状态变更。用单独的一章中探讨更高级的实体映射的一个原因在于，我们认为它们中不少都罕有使用，或者至少是非强制使用的。仅使用组件映射和多对一(偶尔使用一对一)实体关联是可行的。甚至可以在不映射一个集合的情况下编写复杂的应用程序！第 7 章已经介绍了能从集合映射中获得的具体好处；用于适合集合映射的规则也适用于本章中的所有示例。在尝试复杂集合映射之前，应该总是确保真的需要一个集合。

我们从不涉及集合的映射开始：一对一实体关联。

JPA 2 中主要的新功能

- 多对一和一对一关联现在可以使用中间联结/链接表来映射。
- 可嵌入组件类可以具有对实体的单向关联，甚至是具有集合的多值。

8.1 一对一关联

在 5.2 节中论证过，User 和 Address 之间的关系(用户有 billingAddress、homeAddress 和 shippingAddress)最好用@Embeddable 组件映射来表示。这通常是表示一对一关系的最简单方式，因为在这种情况下其生命周期通常具有依赖性。它在 UML 中要么是聚合，要么是组合。

使用一个专用 ADDRESS 表并且将 User 和 Address 都映射为实体又会如何呢？这个模型的一个好处是可以共享引用——另一个实体类(如 Shipment)也可以有对特定 Address 实例的引用。如果 User 也有对这个实例的引用，如它们的 shippingAddress，那么 Address 实例必须支持共享的引用并且需要其自己的标识。

在这种情况下，User 和 Address 类就有了真正的一对一关联。看看图 8-1 中修订后的

类图。

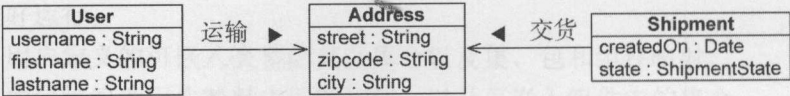


图 8-1 作为实体的 Address 有两个关联，支持共享引用

对于一对一关联，有几种可能的映射。要考虑的第一个策略是共享主键值。

8.1.1 共享主键

通过主键关联联系在一起的两个表中的行会共享相同的主键值。User 具有与其(运输)Address 相同的主键值。这个方法的主要难点在于，要确保在保存所关联的实例时为实例指定相同主键值。查看此问题之前，先创建基本映射。现在 Address 类是一个独立的实体；不再是一个组件，见代码清单 8.1。

代码清单8.1 作为独立实体的Address类

路径: /model/src/main/java/org/jpwh/model/associations/onetooone/sharedprimarykey/Address.java

```
@Entity
public class Address {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String street;

    @NotNull
    protected String zipcode;

    @NotNull
    protected String city;
    // ...
}
```

User 类也是实体，具有 shippingAddress 关联属性，见代码清单 8-2。

代码清单8.2 User实体和shippingAddress关联

路径: /model/src/main/java/org/jpwh/model/associations/onetooone/ sharedprimarykey/ User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @Id
    protected Long id;
```

← 使用应用程序指定的标识符值

```
@OneToOne (
    fetch = FetchType.LAZY,
    optional = false
)
@PrimaryKeyJoinColumn
protected Address shippingAddress;

protected User() {
}

public User(Long id, String username) {
    this.id = id;
    this.username = username;
}
// ...
}
```

①默认为 EAGER

②将实体值属性标记为一对一关联

③使用代理的延迟加载所必须的

④选择共享主键策略

⑤需要标识符

对于 User，没有声明标识符生成器①。正如 4.2.4 小节中所提及的，这在使用应用程序指定的标识符值时很少发生。可以看到该构造函数设计(微弱地)强制实现了这一点⑥：该类的公共 API 需要标识符值来创建实例。

该示例中出现了两个新注解。@OneToOne②会执行所期望的：它要求将实体值类型标记为一对一关联。像往常一样，应该选择延迟加载策略，因此要用 LAZY 重写默认的 FetchType.EAGER③。第二个新的注解是@PrimaryKeyJoinColumn④，选择想要映射的共享主键策略。现在这是从 User 到 Address 的单向共享主键一对一关联映射。

optional=false 切换④会定义 User 必须有 shippingAddress。Hibernate 生成的数据库架构用外键约束反映这一点。USERS 表的主键还有一个引用 Address 表主键的外键约束。参见图 8-2 中的表。

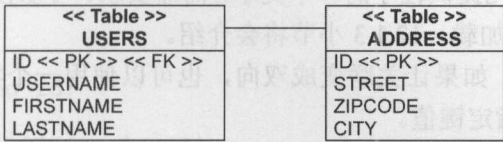


图 8-2 USERS 表在其主键上具有一个外键约束

JPA 规范不包含处理共享主键生成这个问题的标准方法。这意味着你要负责在保存 User 实例的标识符值之前将它正确设置为链接 Address 实例的标识符值：

```
路径: /examples/src/test/java/org/jpwh/test/associations/ OneToOneSharedPrimaryKey.
java

Address someAddress =
    new Address("Some Street 123", "12345", "Some City");

em.persist(someAddress);

User someUser =
    new User(
        someAddress.getId(),
        "johndoe"
    )
```

生成标识符值

指定相同的标识符值


```

    );
    em.persist(someUser);
    someUser.setShippingAddress(someAddress);    ← 可选

```

在持久化 Address 之后，要使用其生成的标识符值并且也要在保存这个值之前在 User 上设置它。这个示例的最后一行是可选的：现在在调用 `someUser.getShippingAddress()` 时，代码会期望一个值，因此应该设置它。如果忘记了这最后一步，Hibernate 也不会返回错误。使用该映射和代码有 3 个问题：

- 必须记住，必须首先保存 Address，然后在调用 `persist()` 之后得到其标识符值。如果在 INSERT 之前 Address 实体有在 `persist()` 上生成值的标识符生成器，那么这就是唯一可行的方法，正如在 4.2.5 小节中所探讨过的。否则，`someAddress.getId()` 会返回 null，并且不能手动设置该 User 的标识符值。
- 使用代理的延迟加载仅在关联不可选的情况下才有效。这通常对于刚接触 JPA 的开发人员来说很意外。`@OneToOne` 默认为 `FetchType.EAGER`：当 Hibernate 加载 User 时，它会立即加载 `shippingAddress`。从概念上将，使用代理的延迟加载仅在 Hibernate 清楚存在链接的 `shippingAddress` 时才合理。如果属性是可为空的，则 Hibernate 就必须通过查询 ADDRESS 表在数据库中检查该属性值是否为 NULL。如果必须检查数据库，那么也可以立即加载值，因为使用代理不会有什么好处。
- 一对一关联是单向的；有时候需要双向导航。

第一个问题没有其他解决方案，并且它是应该总是选择能够在所有 SQL INSERT 之前生成值的标识符生成器的原因之一。

`@OneToOne(optional=true)` 关联不支持使用代理的延迟加载。这与 JPA 规范一致。`FetchType.LAZY` 是持久化提供程序的一个提示，而非要求。可以使用字节码指令得到可为空的 `@OneToOne` 的延迟加载，12.1.3 小节将会介绍。

对于最后一个问题，如果让关联变成双向，也可以使用一个特殊的仅用于 Hibernate 的标识符生成器来帮助指定键值。

8.1.2 外主键生成器

双向映射总是需要 `mappedBy` 侧。此处，选取 User 侧(这是个人喜好，也可能是其他次级需求方面的问题)：

```

路径: /model/src/main/java/org/jpwh/model/associations/onetoone/foreigngenerator/
      User.java

```

```

@Entity
@Table(name = "USERS")
public class User {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @OneToOne(

```

```

        mappedBy = "user",
        cascade = CascadeType.PERSIST
    )
    protected Address shippingAddress;
    // ...
}

```

将其与前一个映射进行对比：添加了 `mappedBy` 选项，告知 Hibernate 低级别的详情现在可以通过名称为 `user` 的“另一侧的属性”来映射。为了方便起见，要启用 `CascadeType.PERSIST`；传递持久化将让其更易于按正确顺序保存实例。让 `User` 持久化时，Hibernate 会将 `shippingAddress` 持久化并且为主键自动生成标识符。

接下来，看看“另一侧”：`Address`，见代码清单 8.3。

代码清单8.3 Address具有特殊的外键生成器

路径：/model/src/main/java/org/jpwh/model/associations/onetooone/foreigngenerator/
Address.java

```

@Entity
public class Address {

    @Id
    @GeneratedValue(generator = "addressKeyGenerator")
    @org.hibernate.annotations.GenericGenerator(
        name = "addressKeyGenerator",
        strategy = "foreign",
        parameters =

            @org.hibernate.annotations.Parameter(
                name = "property", value = "user"
            )
    )
    protected Long id;

    @OneToOne(optional = false)
    @PrimaryKeyJoinColumn
    protected User user;

    protected Address() {

    }

    public Address(User user) {
        this.user = user;
    }

    public Address(User user, String street, String zipcode, String city) {
        this.user = user;
        this.street = street;
        this.zipcode = zipcode;
        this.city = city;
    }
}

```

①定义主键值生成器

②创建外键约束

③Address 必须具有对 User 的引用

④Address 的公共构造函数

```
// ...  
}
```

这是相当多的新代码。先从标识符属性开始，然后处理一对一关联。

Hibernate 特性

标识符属性上的@GenericGenerator❶定义了一个用于特殊目的的主键值生成器，该生成器使用仅限 Hibernate 的外键策略。4.2.5 节的概述中没有介绍这个生成器；共享主键一对一关联是其唯一应用场景。持久化 Address 的实例时，这一特殊生成器会抓取 user 属性的值，并使用所引用的实体实例 User 的标识符值。

继续处理@OneToOne 映射❷。使用@PrimaryKeyJoinColumn 注解将 user 属性标记为共享主键实体关联❸。它被设置为 optional=false，所以 Address 必须有对 User 的引用。Address 的公共构造函数❹现在需要 User 实例。反映 optional=false 的外键约束现在位于 ADDRESS 表的主键列上，可以在图 8-3 中看到该架构。

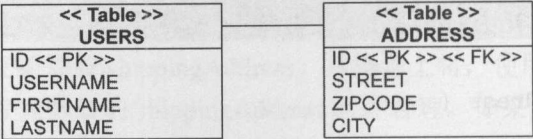


图 8-3 ADDRESS 表在其主键上有一个外键约束

不再必须在工作单元中调用 omeAddress.getId()或 someUser.getId()了。储存数据被简化了：

```
路径: /examples/src/test/java/org/jpwh/test/associations/OneToOneForeignGenerator.  
java
```

```
User someUser = new User("johndoe");  
  
Address someAddress =  
    new Address(  
        someUser,  
        "Some Street 123", "12345", "Some City"  
    );  
  
someUser.setShippingAddress(someAddress);  
em.persist(someUser);
```

链接

shippingAddress 的传递持久化

不要忘记必须链接双向实体关联的两侧。注意，使用这个映射，不会得到 User#shippingAddress 的延迟加载(它是可选/可为空的)，但可以按需使用代理加载 Address#user(它是非可选的)。

共享主键一对一关联相对较少。相反，通常会使用一个外键列和唯一约束来映射“对一”关联。

8.1.3 使用一个外键联结列

相较于共享主键，两个行可以具有基于简单附加外键列的关系。一个表具有引用关联表主键的外键列。(这个外键约束的源和目标甚至可以是相同的表：称之为自引用关系。)

修改 `User#shippingAddress` 的映射。不使用共享主键，现在可以在 `USERS` 表中添加 `SHIPPINGADDRESS_ID` 列。此外，这个列具有 `UNIQUE` 约束，因此相同的运输地址只能被一个用户引用。看看图 8-4 中的架构。

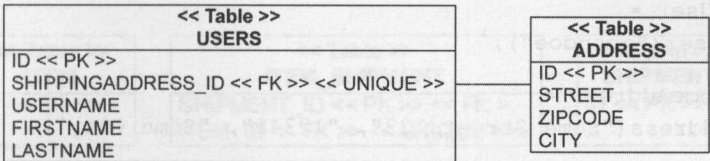


图 8-4 USERS 和 ADDRESS 表之间的一对一联结列关联

`Address` 是一个普通的实体类，就像在本章代码清单 8.1 中所介绍的。`User` 实体类有 `shippingAddress` 属性，实现了这个单向关联：

```
路径: /model/src/main/java/org/jpwh/model/associations/onetooone/foreignkey/
User.java
```

```
@Entity
@Table(name = "USERS")
public class User {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @OneToOne(
        fetch = FetchType.LAZY,
        optional = false,
        cascade = CascadeType.PERSIST
    )
    @JoinColumn(unique = true)
    protected Address shippingAddress;
    // ...
}
```

非空
默认为
SHIPPINGADDRESS_ID

不需要任何特殊的标识符生成器或主键分配；不使用 `@PrimaryKeyJoinColumn`，可以应用常规 `@JoinColumn`。如果对 SQL 比 JPA 更熟悉，则有助于每次看见映射中的 `@JoinColumn` 时想起“外键列”。

应该为这个关联启用延迟加载。不同于共享主键，此处使用延迟加载没有任何问题：当 `USERS` 表的一行已经被加载，它会包含 `SHIPPINGADDRESS_ID` 列的值。因此 `Hibernate` 知道是否出现了 `ADDRESS` 行，并且可以使用代理按需加载 `Address` 实例。

不过，在该映射中，要设置 `optional=false`，这样用户就必须有一个运输地址。这不会影响加载行为，但它是在 `@JoinColumn` 上设置 `unique=true` 的逻辑结果。这一设置会将唯一

约束添加到所生成的 SQL 架构。如果 SHIPPINGADDRESS_ID 列的值必须对于所有用户都是唯一的，那么只有一个用户能够不具有“运输地址”。因此，可为空的唯一列通常没什么意义。

创建、链接和存储实例都很简单明了：

路径：/examples/src/test/java/org/jpwh/test/associations/ OneToOneForeignKey.
java

```
User someUser =
    new User("johndoe");

Address someAddress =
    new Address("Some Street 123", "12345", "Some City");

someUser.setShippingAddress(someAddress);    ←—— 链接
em.persist(someUser);                        ←—— shippingAddress 的传递持久化
```

现在已经完成了两个基本一对一关联映射：第一个有共享主键，第二个有外键引用和唯一列约束。要探讨的最后一个选项会更为奇异一些：借助额外表的帮助来映射一对一关联。

8.1.4 使用一个联结表

你大概已经注意到可为空的列会出现问题。有时候可选值更好的解决方案是中间表，如果出现了一个链接，它就会包含一行，反之则不会。

考虑一下 CaveatEmptor 中的 Shipment 实体并探讨其目的。卖家和买家在 CaveatEmptor 中通过开启拍卖和在拍卖上出价来交互。运输商品看起来超出了该应用程序的范围；卖家和买家会在拍卖结束后就运输和支付方法达成一致。他们可以在 CaveatEmptor 之外的线下交流。

另一方面，可以在 CaveatEmptor 中提供托管服务。卖家会使用该服务在拍卖结束时创建可追踪的运输。买家会向受托人(也就是你)支付该拍卖商品的金额，而你将在资金到位时通知卖家。一旦运输完成并且买家接收商品之后，就要将该笔资金转付给卖家。

如果曾经参与过价值很高的在线拍卖，那么大概已经使用过这样的托管服务了。但在 CaveatEmptor 中想要的更多：不仅仅为已完成拍卖提供受信服务，还要允许在 CaveatEmptor 之外，让用户可以为其在拍卖之外进行的交易创建可追踪和可信任的运输。

这一场景要求 Shipment 实体具有对 Item 的可选一对一关联。看看图 8-5 中用于这个域模型类图。

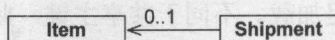


图 8-5 Shipment 具有与拍卖 Item 之间的可选链接

提示：

我们曾简单考虑过不在本节使用 CaveatEmptor 示例，因为无法找到需要可选一对一关联的天然场景。如果这个托管示例看起来不太自然，可以思考下将员工分配到工作站的等

价问题。这也是一个可选的一对一关系。

在该数据库架构中，添加中间链接表 ITEM_SHIPMENT。这个表中的一行就表示拍卖上下文中进行的一个 Shipment。图 8-6 显示了这些表。

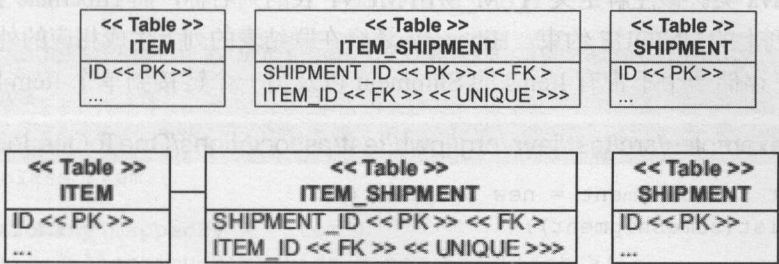


图 8-6 链接商品和运输的中间表

注意该架构是如何强制唯一性和一对一关系的：ITEM_SHIPMENT 的主键是 SHIPMENT_ID 列，而 ITEM_ID 列是唯一的。因此一个商品仅可以位于一次运输中。当然，那也意味着一次运输仅能包含一个商品。

要在 Shipment 实体类中使用 @OneToOne 注解映射这个模型：

```
路径：/model/src/main/java/org/jpwh/model/associations/onetoone/jointable/
Shipment.java
```

```
@Entity
public class Shipment {

    @OneToOne(fetch = FetchType.LAZY)
    @JoinTable(
        name = "ITEM_SHIPMENT",
        joinColumns = {
            @JoinColumn(name = "SHIPMENT_ID"),
            @JoinColumn(name = "ITEM_ID", nullable = false, unique = true)
        },
        inverseJoinColumns = {
            @JoinColumn(name = "SHIPMENT_ID", nullable = false, unique = true)
        }
    )
    protected Item auction;

    public Shipment() {
    }

    public Shipment(Item auction) {
        this.auction = auction;
    }
    // ...
}
```

延迟加载已经被启用，有一点变化：Hibernate 加载一个 Shipment 时，它同时需要 SHIPMENT 和 ITEM_SHIPMENT 联结表。Hibernate 必须在它能使用代理之前知道是否有

一个到 Item 的链接。它会在一个外联结 SQL 查询中这样做,因此不会看到任何额外的 SQL 语句。如果 ITEM_SHIPMENT 中有一行,则 Hibernate 会使用 Item 占位符。

@JoinTable 注解是新的;总是必须指定中间表的名称。这个映射有效隐藏了联结表;没有对应的 Java 类。该注解定义 ITEM_SHIPMENT 表的列名称,而 Hibernate 会在架构中生成 ITEM_ID 列上的 UNIQUE 约束。Hibernate 还会在联结表的列上生成相应的外键约束。

这里你要存储一个不带有 Items 的 Shipment 以及另一个链接到单个 Item 的 Shipment:

路径: /examples/src/test/java/org/jpwh/test/associations/OneToOneJoinTable.java

```
Shipment someShipment = new Shipment();
em.persist(someShipment);

Item someItem = new Item("Some Item");
em.persist(someItem);

Shipment auctionShipment = new Shipment(someItem);
em.persist(auctionShipment);
```

这样就完成了一对一关联映射。概括来说,如果两个实体中的一个总是在另一个之前存储并且可以充当主键源,则使用共享主键关联。要在其他所有情况下使用外键关联,并且在一对一关联可选的情况下使用隐藏中间联结表。

现在专注于多元或多值的实体关联,首先探究用于一对多的一些高级选项。

8.2 一对多关联

根据定义,多元实体关联是多个实体引用的集合。在 7.3.2 小节中,映射过其中一个,即一对多关联。一对多关联是一种最重要的实体关联,它涉及一个集合。介绍了这么多,是为了阻止在简单双向多对一/一对多关联可以达到目的的情况下使用更复杂的关联方式。

另外,要记住,如果不愿意,不必映射实体的任何集合;可以总是编写一个显式查询而非通过遍历进行直接访问。如果决定映射多个实体引用的集合,那么有一些选项可用,现在来探讨一些较为复杂的情况。

8.2.1 考虑一对多包

到目前为止,只看到了 Set 上的 @OneToMany, 但为双向一对多关联转而使用一个包映射是可行的。为何会这样做呢?

包具有所有集合的最高性能特征,可以将之用于双向一对多实体关联。默认情况下,在应用程序中首次访问 Hibernate 里的集合时就会加载它们。由于包不必(像列表一样)维持其元素的索引或者(像集一样)检查重复元素,所以可以将新元素添加到包而不会触发加载。如果打算映射一个可能很大的实体引用集合,那么这就是一个重要特性。

另一方面,不能急于同时获取两个包类型的集合:例如,如果一个 Item 的 bids 和 images 是一对多的包,就不能这样做。这不是什么大的损失,因为同时获取两个集合总是会得到一个笛卡尔积;无论集合是不是包、集或列表,都会希望避免此种操作。第 12 章将回过头

来介绍获取策略。总之，可以认为，如果被映射为 `@OneToMany(mappedBy = "...")`，那么对于一对多关联而言，包就是最佳的反转集合。

要将一个双向一对多关联映射为一个包，必须在 `Item` 实体中使用 `Collection` 和 `ArrayList` 实现替换 `bids` 集合的类型。`Item` 和 `Bid` 之间关联的映射实质上仍旧没有改变：

路径: `/model/src/main/java/org/jpwh/model/associations/onetomany/ bag/Item.java`

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    public Collection<Bid> bids = new ArrayList<>();

    // ...
}
```

具有其 `@ManyToOne` 的 `Bid` 侧(即“被映射”侧)，甚至是表都与 7.3.1 小节中的相同。包还允许重复元素，而之前映射的集则不允许：

路径: `/examples/src/test/java/org/jpwh/test/associations/OneToManyBag.java`

```
Item someItem = new Item("Some Item");
em.persist(someItem);

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);
someItem.getBids().add(someBid);
em.persist(someBid);

assertEquals(someItem.getBids().size(), 2);
```

← 没有持久化效果！

事实证明，这与本例无关，因为重复意味着已经几次将一个特定引用添加到了相同 `Bid` 实例。你不会在应用程序代码中这样做。不过，即便你将相同引用几次添加到这个集合，Hibernate 也会忽略它。与数据库更新相关的一侧是 `@ManyToOne`，并且关系已经被该侧“所映射”了。加载该 `Item` 时，集合不会包含重复项：

路径: `/examples/src/test/java/org/jpwh/test/associations/OneToManyBag.java`

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(item.getBids().size(), 1);
```

正如提到过的，包的优势在于，添加一个新元素时不必初始化集合：

路径: `/examples/src/test/java/org/jpwh/test/associations/OneToManyBag.java`

```
Item item = em.find(Item.class, ITEM_ID);

Bid bid = new Bid(new BigDecimal("456.00"), item);
item.getBids().add(bid);
em.persist(bid);
```

← 没有 SELECT！

这一代码示例会触发一个 SQL 的 SELECT 来加载 Item。如果使用 `em.getReference()` 而非 `em.find()`，那么 Hibernate 仍旧会初始化并且在调用 `item.getBids()` 时立即使用 SELECT 返回一个 Item 代理。但只要不遍历该 Collection，就不需要更多的查询，并且会得到用于新 Bid 的 INSERT，而不会加载所有的 bids。如果该集合是 Set 或 List，则 Hibernate 会在添加另一个元素时加载所有的元素。

将这个集合修改成一个持久化 List。

8.2.2 单向和双向列表映射

如果需要真正的列表来保存集合中元素的位置，则必须在一个额外的列中存储该位置。对于一对多映射，这也意味着应该将 `Item#bids` 属性改为 List 并且用 ArrayList 初始化变量：

路径：/model/src/main/java/org/jpwh/model/associations/onetomany/list/Item.java

```
@Entity
public class Item {

    @OneToMany
    @JoinColumn(
        name = "ITEM_ID",
        nullable = false
    )
    @OrderColumn(
        name = "BID_POSITION",
        nullable = false
    )
    public List<Bid> bids = new ArrayList<>();
    // ...
}
```

这是一个单向映射：没有其他“被映射”侧。Bid 没有 `@ManyToOne` 属性。持久化列表索引需要新注解 `@OrderColumn`，像往常一样，此处应该让列 NOT NULL。图 8-7 中显示了 BID 表的数据库视图，具有联结和顺序列。

BID			
ID	ITEM_ID	BID_POSITION	AMOUNT
1	1	0	99.00
2	1	1	100.00
3	1	2	101.00
4	2	0	4.99

图 8-7 BID 表

每个集合的存储索引从零开始并且是连续的(不存在间隔)。在添加、移除和移动 List 的元素时, Hibernate 会潜在执行许多 SQL 语句。7.1.6 小节中探讨过这一性能问题。

将这个映射变成双向, 在 Bid 实体上使用 @ManyToOne 属性:

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/list/Bid.java

@Entity

```
public class Bid {
```

```
    @ManyToOne
```

```
    @JoinColumn(
```

```
        name = "ITEM_ID",
```

```
        updatable = false, insertable = false
```

← 禁止写入!

```
)
```

```
    @NotNull
```

```
    protected Item item;
```

← 用于架构生成

```
    // ...
```

```
}
```

大概想要不同的代码——可能是 @ManyToOne(mappedBy="bids") 并且没有额外的 @JoinColumn 注解。但 @ManyToOne 没有 mappedBy 属性: 它总是关系的“所有”侧。必须让另一侧 @OneToMany 成为 mappedBy 侧。此处会遇到一个概念问题和一些 Hibernate 奇怪之处。

Item#bids 集合不再是只读的, 因为 Hibernate 现在必须存储每个元素的索引。如果 Bid#item 侧是关系的所有者, 那么在存储数据并且不写入元素索引时, Hibernate 就会忽略该集合。必须映射 @JoinColumn 两次, 然后使用 updatable=false 和 insertable=false 在 @ManyToOne 侧禁止写入。现在 Hibernate 就会在存储数据时考虑集合侧包含每个元素索引的问题。@ManyToOne 实际上是只读的, 就像它有 mappedBy 属性一样。

具有 mappedBy 的双向列表

Hibernate 上开放了与此问题有关的一些已知缺陷报告; 将来的版本可能会允许使用集合上 @OneToMany(mappedBy) 和 @OrderColumn 的兼容 JPA。在编写本书时, 所示的映射是唯一可用于具有持久化列表的双向一对多的有效变体。

最后, Hibernate 架构生成器总是依赖于 @ManyToOne 侧的 @JoinColumn。因此, 如果希望产生正确的架构, 就应该在这一侧添加 @NotNull 或者声明 @JoinColumn(nullable=false)。如果有 @ManyToOne, 生成器会忽略 @OneToMany 侧及其联结列。

在真实的应用程序中, 不会用 List 映射关联。保存数据库中元素的顺序看起来像是常见用例, 但重新考虑一下, 它又并非十分有用: 有时候希望首先显示具有最高或最新出价的列表, 或者仅显示指定用户的出价, 或者仅显示特定时间段的出价。这些操作都不需要连续的列表索引。正如 3.2.4 小节中所提及的, 要避免在数据库中存储显示顺序; 要用查询而非硬编码映射保持其灵活性。此外, 当应用程序移除、添加或移动列表中的元素时, 维护索引的代价将会很高, 并且可能触发许多 SQL 语句。应使用 @ManyToOne 映射该外

键联结列，并且删除该集合。

接下来是具有有一对多关系的另一个场景：映射到中间联结表的关联。

8.2.3 具有联结表的可选一对多

Item 类的一个有用附加是 buyer 属性。然后可以调用 someItem.getBuyer()来访问中标的 User。如果让其变成双向，则这个关联还将有助于呈现显示特定用户中标的所有拍卖的界面：要调用 someUser.getBoughtItems()而不是编写查询。

从 User 类的角度看，该关联是一对多的。图 8-8 显示了这些类及其关系。



图 8-8 User-Item “购买”关系

为何这个关联与 Item 和 Bid 之间的关联不同呢？UML 中的多样性 0..*表明，该引用是可选的。这不会过多影响 Java 域模型，但它对底层表有影响。ITEM 表中应该有一个 BUYER_ID 外键列。该列必须是可为空的，因为用户可能没有购买特定 Item(只要拍卖仍在进行)。

可以接受外键列可为 NULL 并且应用额外的约束：“仅在未到达拍卖结束时间或没有出价的情况下才允许为 NULL。”在关系型数据库架构中，总是试图避免可为空的列。未知的信息会降低存储的数据的质量。元组表示为真的命题；不能断言不知道的东西。此外，实际上许多开发人员和 DBA 都不会创建正确的约束，并且常会依赖有问题的应用程序代码来提供数据完整性。

可选的实体关联，如一对一或一对多，在 SQL 数据库中用一个联结表来表示是最合适的。图 8-9 显示了一个示例架构。

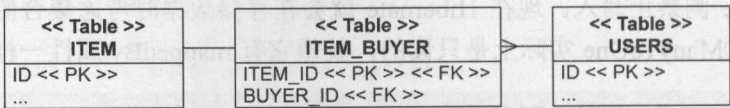


图 8-9 一个中间表链接用户和商品

在本章前面的内容中为一对一关联添加了联结表。要保证一对一的多样性，需要在联结表的外键列上应用唯一约束。在当前的例子中，有一对多的多样性，所以只有 ITEM_ID 主键列必须是唯一的：每一次只有一个 User 可以购买任何指定 Item。BUYER_ID 列不是唯一的，因为一个 User 可以购买许多 Items。

User#boughtItems 集合的映射很简单：

```
路径: /model/src/main/java/org/jpwh/model/associations/onetomany/jointable/
User.java

@Entity
@Table(name = "USERS")
public class User {

    @OneToMany(mappedBy = "buyer")
    protected Set<Item>boughtItems = new HashSet<Item>();
}
```

```
// ...
}
```

这是双向关联常见的只读侧，在“被映射”侧 `Item#buyer` 上具有对架构的实际映射：

路径：/model/src/main/java/org/jpwh/model/associations/onetomany/jointable/
Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinTable(
        name = "ITEM_BUYER",
        joinColumns =
            @JoinColumn(name = "ITEM_ID") ← 默认为 ID
        inverseJoinColumns =
            @JoinColumn(nullable = false) ← 默认为 BUYER_ID
    )
    protected User buyer;

    // ...
}
```

现在这是一个干净、可选的一对多/多对一关系。如果一个 `Item` 还没有被购买，则联结表 `ITEM_BUYER` 中就没有对应的行。架构中没有任何会出问题的可为空列。然而，应该为 `ITEM_BUYER` 表编写一个在 `INSERT` 上运行的程序性约束和触发器：“仅在已经到了给定商品的拍卖结束时间并且用户中标的情况下才允许插入一个买家。”

接下来的示例是使用一对多关联的最后一个示例。到目前为止，你已经看到了从一个实体到另一个实体的一对多关联。可嵌入组件类也可以具有对实体的一对多关联。

8.2.4 可嵌入类中的一对多关联

再次思考已经为一些章节重复使用过的可嵌入组件映射：`User` 的 `Address`。现在要通过添加从 `Address` 到 `Shipment` 的一对多关联来扩展这个示例：集合 `deliveries`。图 8-10 显示了用于这个模型的 UML 类图。

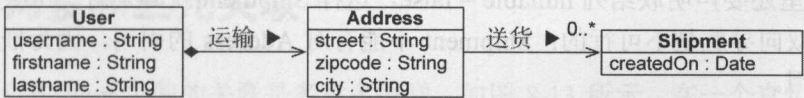


图 8-10 从 `Address` 到 `Shipment` 的一对多关系

`Address` 是一个 `@Embeddable` 类，而不是实体。它可以拥有到一个实体的单向关联；此处它是到 `Shipment` 的一对多多样性。（8.3 节将介绍具有多对一关联的一个可嵌入类。）

`Address` 类具有表示这一关联的 `Set<Shipment>`：

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/embeddable/
Address.java

```
@Embeddable
public class Address {

    @NotNull
    @Column(nullable = false)
    protected String street;

    @NotNull
    @Column(nullable = false, length = 5)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String city;

    @OneToMany
    @JoinColumn(
        name = "DELIVERY_ADDRESS_USER_ID", ← 默认为 DELIVERIES_ID
        nullable = false
    )
    protected Set<Shipment> deliveries = new HashSet<Shipment>();
    // ...
}
```

用于这个关联的第一个映射策略是使用名称为 DELIVERY_ADDRESS_USER_ID 的 @JoinColumn。这个外键约束的列位于 SHIPMENT 表中，如图 8-11 所示。

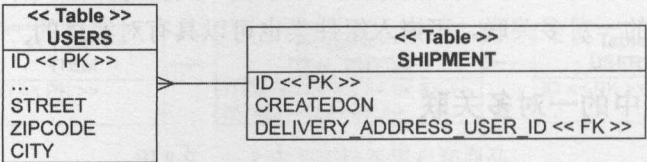


图 8-11 USERS 表中的主键链接了 USERS 和 SHIPMENT 表

可嵌入组件没有其自己的标识符，因此外键列中的值就是 User 的标识符值，它嵌入了 Address。这里还要声明联结列 nullable = false，这样 Shipment 就必须有一个关联的送货地址。当然，双向导航是不可行的：Shipment 不能有对 Address 的引用，因为嵌入的组件不能有共享引用。

如果该关联是可选的并且不希望使用可为空的列，可以将该关联映射到中间联结/链接表，如图 8-12 所示。Address 中集合的映射现在会使用 @JoinTable 而非 @JoinColumn：

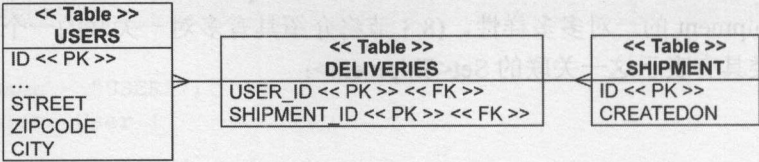


图 8-12 在 USERS 和 SHIPMENT 之间使用一个中间表来表示可选关联

路径: /model/src/main/java/org/jpwh/model/associations/onetomany/ embeddablejointable/
Address.java

```
@Embeddable
public class Address {

    @NotNull
    @Column(nullable = false)
    protected String street;

    @NotNull
    @Column(nullable = false, length = 5)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String city;

    @OneToMany
    @JoinTable(
        name = "DELIVERIES",           ← 默认为 USERS_SHIPMENT
        joinColumns =
        @JoinColumn(name = "USER_ID"), ← 默认为 USERS_ID
        inverseJoinColumns =
        @JoinColumn(name = "SHIPMENT_ID") ← 默认为 SHIPMENTS_ID
    )
    protected Set<Shipment> deliveries = new HashSet<Shipment>();
    // ...
}
```

注意，如果不声明@JoinTable 或@JoinColumn，则可嵌入类中的@OneToMany 就会默认为联结表策略。

从所属实体类中，可以使用@AttributeOverride 重写可嵌入类的属性映射，如 5.2.3 小节中所介绍的那样。如果希望重写可嵌入类中实体关联的联结表或列映射，则要转而在所属实体类中使用@AssociationOverride。然而，不能切换映射策略；可嵌入组件类中的映射会决定使用联结表还是联结列。

联结表映射当然也适用于真正的多对多映射。

8.3 多对多和三元关联

Category 和 Item 之间的关联是多对多关联，如图 8-13 所示。在一个真实系统中，可能没有多对多关联。我们的经验是，关联实体之间的每个链接都几乎总是必须附加上其他信息。例如将 Item 添加到负责创建链接的 Category 和 User 时需要的时间戳。本节稍后的内容中将扩展该示例来介绍这样一个例子。应该从常规且较简单的多对多关联开始。

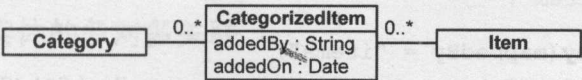


图 8-13 Category 和 Item 之间的多对多关联

8.3.1 单向和双向多对多关联

数据库中的联结表表示常规的多对多关联，有些开发人员也将其称为链接表或关联表。图 8-14 显示了具有链接表的多对多关系。

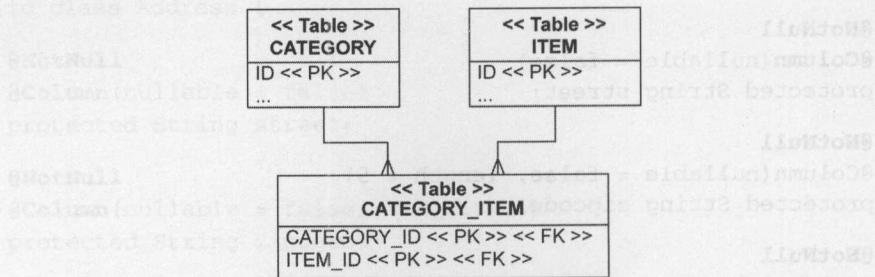


图 8-14 具有链接表的多对多关系

链接表 CATEGORY_ITEM 有两列，都分别具有引用 CATEGORY 和 ITEM 表的外键约束。其主键是这两列的组合键。只能链接某个特定 Category 和 Item 一次，但可以将相同商品链接到几个类别。

在 JPA 中，可以在集合上使用 @ManyToMany 映射多对多关联：

路径: /model/src/main/java/org/jpwh/model/associations/ manytomany/bidirectional/ Category.java

```
@Entity
public class Category {

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID")
    )
    protected Set<Item> items = new HashSet<Item>();

    // ...
}
```

像往常一样，可以启用 CascadeType.PERSIST 来让它更易于保存数据。当从该集合引用新 Item 时，Hibernate 会让其持久化。让这一关联变成双向(如果不需要，则不必这么做)：

路径: /model/src/main/java/org/jpwh/model/associations/manytomany/ bidirectional/ Item.java

```
@Entity
public class Item {

    @ManyToMany(mappedBy = "items")
    protected Set<Category> categories = new HashSet<Category>();
}
```



```
// ...  
}
```

就像所有双向映射一样，一侧是由另一侧“映射的”。Item#categories 集合是只读的；Hibernate 将在存储数据时分析 Category#items 侧的内容。接着要创建两个类别和两个商品，并且用多对多多样性来链接它们：

路径: /examples/src/test/java/org/jpwh/test/associations/ ManyToManyBidirectional.
java

```
Category someCategory = new Category("Some Category");  
Category otherCategory = new Category("Other Category");  
  
Item someItem = new Item("Some Item");  
Item otherItem = new Item("Other Item");  
  
someCategory.getItems().add(someItem);  
someItem.getCategories().add(someCategory);  
  
someCategory.getItems().add(otherItem);  
otherItem.getCategories().add(someCategory);  
  
otherCategory.getItems().add(someItem);  
someItem.getCategories().add(otherCategory);  
  
em.persist(someCategory);  
em.persist(otherCategory);
```

由于启用了传递持久化，所以保存类别会得到实例持久化的整个网络。另一方面，串联选项 ALL、REMOVE 以及孤立的删除(参阅 7.3.3 小节)对于多对多关联没什么意义。这是验证是否理解实体和值类型的好机会。尝试用合理的答案来回答为何这些串联类型对于多对多关联没有意义。

能用 List 代替 Set，甚或包吗？Set 与数据库架构的匹配很完美，因为 Category 和 Item 之间不能有重复链接。

包意味着重复元素，因此需要将不同的主键用于该联结表。Hibernate 的专有@CollectionId 注解可以提供这一功能，如 7.1.5 小节所述。如果需要支持重复链接，稍后将探讨的一种可选多对多策略会是一种更好的选择。

可以使用常规的@ManyToMany 映射像 List 这样的索引集合，但只能在一侧使用。记住，在双向关系中，一侧必须被另一侧“映射”，这意味着当 Hibernate 与数据库同步时它的值会被忽略。如果两侧都是列表，则只能让一侧的索引持久化。

常规的@ManyToMany 映射会隐藏该链接表；没有对应的 Java 类，只有一些集合属性。因此无论何时有人说，“我的链接表具有关于该链接信息的更多列”——并且，根据我们的经验，有些人迟早总是会说出这句话——那就需要将这一信息映射到 Java 类。

8.3.2 具有中间实体的多对多关联

可能总是会将多对多关联表示成一个中介类的两个多对一关联。不隐藏链接表，而

是使用一个 Java 类来表示它。这个模型通常更易于扩展，因此倾向于在应用程序中使用常规多对多关联。在稍后不可避免地要将更多类添加到一个链接表时，变更代码就需要大量的工作；所以在如 8.3.1 小节一样映射@ManyToMany 之前，应该考虑一下图 8-15 中所示的备选方案。

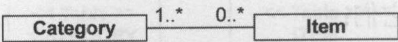


图 8-15 CategorizedItem 是 Category 和 Item 之间的链接

想象一下，需要在每次将 Item 添加到 Category 时记录一些信息。CategorizedItem 会捕获时间戳以及创建该链接的用户。这个域模型需要联结表上有更多的列，如图 8-16 所示。

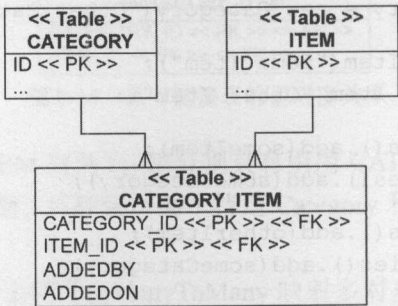


图 8-16 多对多关系中联结表上的附加列

这个新的 CategorizedItem 实体映射到了该链接表，如代码清单 8.4 所示。

代码清单8.4 用CategorizedItem映射多对多关系

路径: /model/src/main/java/org/jpwh/model/associations/manytomany/linkentity/
CategorizedItem.java

```
@Entity
@Table(name = "CATEGORY_ITEM")
@org.hibernate.annotations.Immutable
public class CategorizedItem {

    @Embeddable
    public static class Id implements Serializable {

        @Column(name = "CATEGORY_ID")
        protected Long categoryId;

        @Column(name = "ITEM_ID")
        protected Long itemId;

        public Id() {
        }

        public Id(Long categoryId, Long itemId) {
            this.categoryId = categoryId;
            this.itemId = itemId;
        }

        public boolean equals(Object o) {
            if (o != null && o instanceof Id) {
```

← ①声明类不可变

②封装组合键

```

        Id that = (Id) o;
        return this.categoryId.equals(that.categoryId)
            && this.itemId.equals(that.itemId);
    }
    return false;
}
public int hashCode() {
    return categoryId.hashCode() + itemId.hashCode();
}
}

// ③ 映射标识符属性和组合键列
@Entity
@EmbeddedId
protected Id id = new Id();

@Column(uptdatable = false)
@NotNull
protected String addedBy; // ④ 映射用户名

@Column(uptdatable = false)
@NotNull
protected Date addedOn = new Date(); // ⑤ 映射时间戳

@ManyToOne
@JoinColumn(
    name = "CATEGORY_ID",
    insertable = false, updatable = false)
protected Category category; // ⑥ 映射类别

@ManyToOne
@JoinColumn(
    name = "ITEM_ID",
    insertable = false, updatable = false)
protected Item item; // ⑦ 映射商品

public CategorizedItem(
    String addedByUsername, // ⑧ 构造 CategorizedItem
    Category category,
    Item item) {

    this.addedBy = addedByUsername;
    this.category = category;
    this.item = item;

    this.id.categoryId = category.getId();
    this.id.itemId = item.getId();

    category.getCategorizedItems().add(this);
    item.getCategorizedItems().add(this);
}
// ...
}

```

设置字段

设置标识符值

确保变成双向时的引用完整性

这是具有一些新注解的一大块代码。首先，它是不可变的实体类，因此永远不能在创

建之后更新属性。如果声明这个类为不可变❶，则 Hibernate 可以做一些优化，例如在刷新持久化上下文期间避免脏检查。

实体类需要一个标识符属性。该链接表的主键是 CATEGORY_ID 和 ITEM_ID 的组合。因此，该实体类也有组合键，为方便起见可以在静态嵌套的可嵌入组件类❷中封装它。当然，可以将这个类外部化到其自己的文件中。新的 @EmbeddedId 注解❸会将标识符属性及其组合键列映射到该实体的表。

接下来是两个将 addedBy 用户名❹和 addedOn 时间戳❺映射到联结表的列的基本属性。这是要关注的“关于该链接的附加信息”。

之后两个 @ManyToOne 属性，类别❻和商品❼会映射已经在标识符中映射过的列。这里的技巧是，使用 updatable=false、insertable=false 设置让它们为只读。这意味着 Hibernate 会通过使用 CategorizedItem 的标识符值来写入这些列的值。同时，可以分别通过 categorizedItem.getItem() 和 getCategory() 来读取和浏览所关联的实例。(如果映射相同的列两次，而不将映射变成只读，则 Hibernate 会在启动时提示存在重复列映射。)

还可以看到，构造 CategorizedItem❸涉及设置标识符的值——应用程序总是会指定组合键值；Hibernate 不会生成它们。要特别注意该构造函数以及它如何设置字段值并且通过管理关联两侧的集合来确保引用完整性。之后可以映射这些集合来启用双向导航。

这是一个单向映射，并且足以支持 Category 和 Item 之间的多对多关系。要创建一个链接，要实例化并且持久化一个 CategorizedItem。如果想要打破一个链接，则可以移除该 CategorizedItem。CategorizedItem 的构造函数要求提供已经持久化的 Category 和 Item 实例。

如果双向导航是必须的，那么可以在 Category 和/或 Item 中映射 @OneToMany 集合：

路径: /model/src/main/java/org/jpwh/model/associations/manytomany/linkentity/
Category.java

```
@Entity
public class Category {

    @OneToMany(mappedBy = "category")
    protected Set<CategorizedItem> categorizedItems = new HashSet<>();

    // ...
}
```

路径: /model/src/main/java/org/jpwh/model/associations/manytomany/linkentity/
Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    protected Set<CategorizedItem> categorizedItems = new HashSet<>();

    // ...
}
```

这两侧在 CategorizedItem 中都是被这些注解“映射的”，因此在遍历由其中一个

getCategorizedItems()方法返回的集合时, Hibernate 已经知道要做什么了。

下面这是创建和存储链接的方式:

```
路径: /examples/src/test/java/org/jpwh/test/associations/ManyToManyLinkEntity.  
java
```

```
Category someCategory = new Category("Some Category");  
Category otherCategory = new Category("Other Category");  
em.persist(someCategory);  
em.persist(otherCategory);  
  
Item someItem = new Item("Some Item");  
Item otherItem = new Item("Other Item");  
em.persist(someItem);  
em.persist(otherItem);  
  
CategorizedItemlinkOne = new CategorizedItem(  
    "johndoe", someCategory, someItem  
);  
  
CategorizedItemlinkTwo = new CategorizedItem(  
    "johndoe", someCategory, otherItem  
);  
  
CategorizedItemlinkThree = new CategorizedItem(  
    "johndoe", otherCategory, someItem  
);  
  
em.persist(linkOne);  
em.persist(linkTwo);  
em.persist(linkThree);
```

这一策略的主要好处是双向导航的可能性: 可以通过调用 `someCategory.getCategorizedItems()` 来得到类别中的所有商品, 然后还能使用 `someItem.getCategorizedItems()` 进行反向导航。它的缺点在于, 需要更复杂的代码来管理 `CategorizedItem` 实体实例, 以便创建和移除链接, 这些链接必须独立保存和删除。在 `CategorizedItem` 类中还需要一些基础设施, 如组合标识符。在一些关联上启用 `CascadeType.PERSIST` 会是一个小的改进, 这样就能减少 `persist()` 的调用次数。

在前面的示例中, 将创建 `Category` 和 `Item` 之间链接的用户存储为简单的姓名字符串。而如果该联结表有外键列 `USER_ID`, 就会有一种三元关系。`CategorizedItem` 会有一个用于 `Category`、`Item` 和 `User` 的 `@ManyToOne`。

8.3.3 小节另一种多对多策略。为了让它更具相关性, 让它变成三元关联。

8.3.3 具有组件的三元关联

在 8.3.2 小节中, 用一个映射到链接表的实体类表示了多对多关系。一种可能更简单的替代方式是, 映射到可嵌入组件类:

路径: /model/src/main/java/org/jpwh/model/associations/manytomany/ternary/
CategorizedItem.java

```
@Embeddable
public class CategorizedItem {

    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        nullable = false, updatable = false
    )
    protected Item item;

    @ManyToOne
    @JoinColumn(
        name = "USER_ID",
        updatable = false
    )
    @NotNull
    protected User addedBy;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(updatable = false)
    @NotNull
    protected Date addedOn = new Date();

    protected CategorizedItem() {
    }

    public CategorizedItem(User addedBy,
                           Item item) {
        this.addedBy = addedBy;
        this.item = item;
    }

    // ...
}
```

← 未生成 SQL 约束，
因此不是主键的一
部分

这里的这个新映射是@Embeddable 中的@ManyToOne 关联，以及额外的外键联结列 USER_ID，让其变成一种三元关系。看看图 8-17 中的数据库架构。

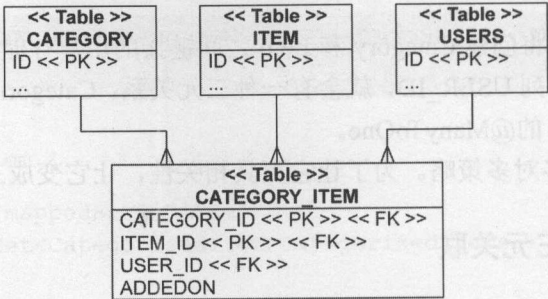


图 8-17 有 3 个外键列的链接表

该可嵌入组件集合的所有者是 Category 实体:

路径: /model/src/main/java/org/jpwh/model/associations/manytomany/ternary/
Category.java

```
@Entity
public class Category {

    @ElementCollection
    @CollectionTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID")
    )
    protected Set<CategorizedItem> categorizedItems = new HashSet<>();

    // ...
}
```

遗憾的是, 这个映射并不完美: 当映射可嵌入类型的 `@ElementCollection` 时, 该目标类型是 `nullable=false` 的所有属性都会变成该(组合)主键的一部分。希望 `CATEGORY_ITEM` 中所有的列都成为 `NOT NULL`。不过, 只有 `CATEGORY_ID` 和 `ITEM_ID` 列应该成为主键的一部分。技巧在于, 将 Bean 验证 `@NotNull` 注解用在不应是主键一部分的属性上。在这种情况下(由于它是一个可嵌入类), Hibernate 会为主键识别和 SQL 架构生成忽略 Bean 验证注解。其缺点是, 所生成的架构在 `USER_ID` 和 `ADDEDON` 列上不会有相应的 `NOT NULL` 约束, 应该手动添加它们。

这个策略的优势是链接组件的隐式生命周期。要创建 `Category` 和 `Item` 之间的关联, 需要将新的 `CategorizedItem` 实例添加到该集合。要打破该链接, 只要从集合中移除该元素即可。这里不需要额外的串联设置, Java 代码也被简化了(尽管分布为多行):

路径: /examples/src/test/java/org/jpwh/test/associations/ManyToManyTernary.
java

```
Category someCategory = new Category("Some Category");
Category otherCategory = new Category("Other Category");
em.persist(someCategory);
em.persist(otherCategory);

Item someItem = new Item("Some Item");
Item otherItem = new Item("Other Item");
em.persist(someItem);
em.persist(otherItem);

User someUser = new User("johndoe");
em.persist(someUser);

CategorizedItem linkOne = new CategorizedItem(
    someUser, someItem
);
someCategory.getCategorizedItems().add(linkOne);

CategorizedItem linkTwo = new CategorizedItem(
    someUser, otherItem
```

```
);
someCategory.getCategorizedItems().add(linkTwo);

CategorizedItem linkThree = new CategorizedItem(
    someUser, someItem
);
otherCategory.getCategorizedItems().add(linkThree);
```

没有办法启用双向导航：根据定义，`CategorizedItem` 这样的可嵌入组件不能使用共享引用。不能从 `Item` 导航到 `CategorizedItem`，并且 `Item` 中没有这个链接的映射。相反，可以编写一个查询来检索指定 `Item` 的类别：

路径：/examples/src/test/java/org/jpwh/test/associations/ManyToManyTernary.java

```
Item item = em.find(Item.class, ITEM_ID);

List<Category> categoriesOfItem =
    em.createQuery(
        "select c from Category c " +
        "join c.categorizedItems ci " +
        "where ci.item= :itemParameter")
        .setParameter("itemParameter", item)
        .getResultList();

assertEquals(categoriesOfItem.size(), 2);
```

现在就完成了首个三元关联映射。在前面几章中，看到了使用映射的 ORM 示例；所介绍的映射的键和值一直是基本或可嵌入类型。8.4 节将介绍更复杂的键/值对类型及其映射。

8.4 具有映射的实体关联

映射键和值可以被引用到其他实体，以便为映射多对多和三元关系提供另一种策略。首先，假设只有每个映射条目的值才是到另一个实体的引用。

8.4.1 具有属性键的一对多关联

如果每个映射条目的值都是到另一个实体的引用，那么就有了一对多实体关系。该映射的键是一个基本类型，如 `Long` 值。

这种结构的一个示例就是有 `Bid` 实例映射的 `Item` 实体，其中每个映射条目都是一对 `Bid` 标识符以及到一个 `Bid` 实例的引用。通过 `someItem.getBids()` 进行遍历时，就是在遍历类似(1, <reference to Bid with PK 1>)、(2, <reference to Bid with PK 2>)的映射条目：

路径：/examples/src/test/java/org/jpwh/test/associations/MapsMapKey.java

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(item.getBids().size(), 2);
```

```
for (Map.Entry<Long, Bid> entry : item.getBids().entrySet()) {  
    assertEquals(entry.getKey(),  
        entry.getValue().getId());  
}
```

← 键是每个 Bid 的标识符

这一映射的底层表并不特殊；你有一个 ITEM 和一个 BID 表，BID 表中有 ITEM_ID 外键列。这与图 7.14 中所示的用于有常规集合而非 Map 的一对多/多对一映射的架构相同。这里动机是应用程序中数据稍有不同的表示形式。

在 Item 类中，包括一个名称为 bids 的 Map 属性：

路径：/model/src/main/java/org/jpwh/model/associations/maps/mapkey/Item.java

@Entity

```
public class Item {
```

```
    @MapKey(name = "id")
```

```
    @OneToMany(mappedBy = "item")
```

```
    protected Map<Long, Bid> bids = new HashMap<>();
```

```
    // ...  
}
```

这里有一个新的@MapKey 注解。它映射了目标实体的属性，在该例中是 Bid 实体作为映射的键。如果省略 name 属性，则默认会是目标实体的标识符属性，这样此处就不需要 name 选项了。因为映射的键会组成一个集，所以对于特定映射，应该期望值是唯一的。这是 Bid 主键的情况，但可能不是 Bid 所有其他属性的情况。确保所选择的属性具有唯一值取决于你——Hibernate 不会检查。

这个映射技术的主要且罕见用例是期望用条目实体值作为条目键来遍历映射条目，这样做的可能原因是对于你希望的呈现数据的方式会很便利。一种更为常见的情形是三元关联中间的映射。

8.4.2 键/值三元关系

现在可能有点烦了，但我们保证这是最后一次介绍另一种映射 Category 和 Item 之间关联的方式。在 8.3.3 小节中，使用了可嵌入 CategorizedItem 组件来表示链接。这里介绍具有 Map 的关系的表示形式，作为附加的 Java 类的替代。每个映射条目的键都是 Item，并且相关的值是将该 Item 添加到 Category 的 User，如图 8-18 所示。

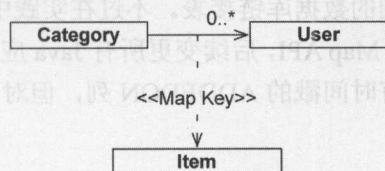


图 8-18 具有实体关联作为键/值对的 Map

正如可以在图 8-19 中所看到的，该架构中的链接/联结表有 3 个列：CATEGORY_ID、ITEM_ID 和 USER_ID。该 Map 是由 Category 实体所有的：

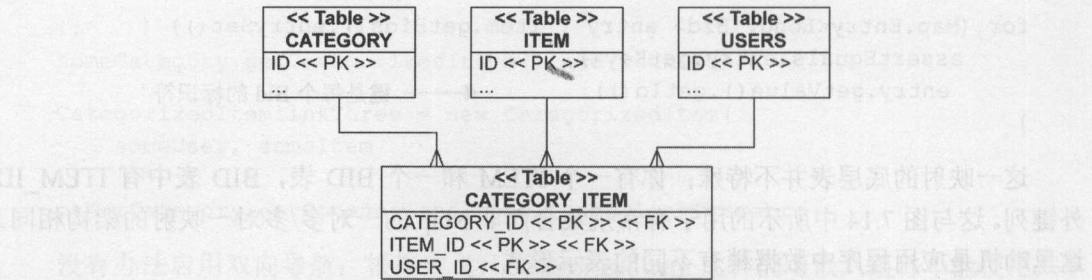


图 8-19 表示 Map 键/值对的链接表

路径: /model/src/main/java/org/jpwh/model/associations/maps/ternary/Category.java

```
@Entity
public class Category {

    @ManyToOne(cascade = CascadeType.PERSIST)
    @MapKeyJoinColumn(name = "ITEM_ID")
    @JoinTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID"),
        inverseJoinColumns = @JoinColumn(name = "USER_ID")
    )
    protected Map<Item, User> itemAddedBy = new HashMap<>();
    // ...
}
```

默认为 ITEMADDED_BY_KEY

@MapKeyJoinColumn 是可选的; Hibernate 会为引用 ITEM 表的联结/外键列默认列名称 ITEMADDED_BY_KEY。

要创建所有 3 个实体之间的链接, 所有的实例都必须已经处于持久化状态, 然后被放入映射中:

路径: /examples/src/test/java/org/jpwh/test/associations/MapsTernary.java

```
someCategory.getItemAddedBy().put(someItem, someUser);
someCategory.getItemAddedBy().put(otherItem, someUser);
otherCategory.getItemAddedBy().put(someItem, someUser);
```

要移除该链接, 需要从映射中移除条目。这对于管理一个复杂关系来说是一个方便的 Java API, 可以隐藏有 3 个列的数据库链接表。不过在实践中要记住, 链接表通常会增加额外的列, 并且如果依赖一个 Map API, 后续变更所有 Java 应用程序代码的代价将会很高。之前在创建链接时使用了带有时间戳的 ADDEDON 列, 但对于这个映射, 必须删除它。

8.5 本章小结

- 介绍了如何使用一对一关联、一对多关联、多对多关联、三元关联以及具有映射的实体关联来映射复杂实体关联。

复杂和遗留架构

9

第 9 章

本章内容简介：

- 用自定义 DDL 改进 SQL 架构
- 集成遗留数据库
- 映射组合键

在本章中我们将专注于系统的最重要部分：数据库架构，其中驻留着完整性规则的集合——创建的现实环境的模型。如果应用程序只能在现实环境中拍卖一个商品，那么数据库架构就应该确保如此。如果一次拍卖总是具有一个起拍价，那么数据库模型就应该包含合适的约束。如果数据满足所有的完整性规则，那么数据就是一致的，我们将在 11.1 节中再次遇到“一致”这个词。

我们还假设该一致性数据是正确的：数据库显式或隐式表述的一切，都是真的；其余一切都是假的。如果希望知道更多有关此方法背后理论的内容，可以查看 *closed-world assumption(CWA)*。

JPA 2 中主要的新功能

- 现在，启动期间的架构生成和自定义 SQL 脚本的执行是标准的，并且可以在持久化单元上进行配置。
- 可以使用标准注解映射并且自定义架构构件，比如索引和外键名称。
- 可以将 `@MapsId` 用作“派生标识”，以便在组合主键中映射外键/多对一关联。

有时可以自上而下启动一个项目。没有已经存在的数据库架构并且可能甚至都没有任何数据——应用程序完全是新的。许多开发人员都喜欢让 Hibernate 自动生成数据库架构的脚本。你大概也会让 Hibernate 在你的开发机器或者用于集成测试的连续构建系统上部署测试数据库的架构。之后，DBA 将使用生成的脚本并且编写用于生产部署的改进后的和最终的架构。本章的第一部分将介绍如何从 JPA 和 Hibernate 中改进架构，以便让你的 DBA 满意。

这方面的另一端是具有现存的、复杂的架构的系统，它具有多年来有价值的数据。新应用程序只是大机器中的一个小齿轮，并且 DBA 不允许对数据库进行任何(有时甚至是无干扰备份)修改。需要一种灵活的对象/关系映射，这样就不必在情况不太对时过多地调整和修改 Java 类。这将是本章后半部分的主题，其中包括组合主键和外键的探讨。

我们先来讨论干净的实现以及 Hibernate 生成的架构。

9.1 改进数据库架构

Hibernate 读取 Java 领域模型类，并且映射元数据和生成架构 DDL 语句。无论何时运行集成测试，都可以将其导出成一个文本文件或者在数据库上直接执行它们。由于架构语言主要是特定于供应商的，因此放入映射元数据的每一个选项都有可能将元数据绑定到特定数据库产品——在使用架构特性时要牢记这一点。

Hibernate 会为表和约束自动创建基本架构；它甚至会根据你选择的标识符生成器创建序列。但有一些架构构件 Hibernate 不能也不会自动创建。其中包括各种高度特定于供应商的性能选项以及只与数据物理存储(例如表空间)有关的所有其他构件。除了这些物理问题外，DBA 通常会提供自定义的附加架构语句来改进所生成的架构。DBA 应该及早介入并且验证从 Hibernate 自动生成的架构。永远不要将未检查的自动生成架构放入生产环境。

如果开发过程允许，DBA 作出的变更就可以回流到 Java 系统中，以便添加到映射元数据。在许多项目中，映射元数据都可以包含来自 DBA 的所有必要架构变更。之后 Hibernate 就可以通过包含所有的注释、约束、索引等在常规构建期间生成最终的生产架构。

在后面几节中，我们将介绍如何自定义生成的架构以及如何添加辅助的数据库架构构件(artifact)(有时我们称之为对象；此处指的不是 Java 对象)。我们将探讨自定义数据类型、额外的完整性规则、索引，以及如何才能替换 Hibernate 生成的一些自动生成的构件名称。

将架构脚本导出到文件

Hibernate 会用一个可以从命令行运行的 `main()` 方法捆绑 `org.hibernate.tool.hbm2ddl.SchemaExport` 类。这个实用工具可以直接与 DBMS 通信并且创建架构或者用 DDL 脚本编写一个文本文件以用于 DBA 的进一步自定义。

首先，我们看看如何才能将自定义 SQL 语句添加到 Hibernate 的自动架构生成过程。

9.1.1 添加辅助数据库对象

可以将以下三种类型的自定义 SQL 脚本挂接到 Hibernate 的架构生成过程中：

- 创建脚本会在架构生成时执行。自定义创建脚本可以在 Hibernate 自动生成的脚本之前、之后运行，或者替代它。换句话说，可以从映射元数据中编写一个在 Hibernate 生成表、约束等之前或之后运行的 SQL 脚本。
- 删除脚本会在 Hibernate 移除架构构件时执行。就像创建脚本一样，删除脚本可以在 Hibernate 自动生成的语句之前、之后运行，或者替代它。
- 加载脚本总是在 Hibernate 生成架构之后运行，作为创建后的最后一步。其主要目的是在应用程序或单元测试运行之前导入测试或主数据。如果希望进一步自定义该架构，那么它可以包含任何种类的 SQL 语句，包括像 ALTER 这样的 DDL 语句。

架构生成过程的这种自定义实际上是标准化的；可以在 `persistence.xml` 中使用 JPA 属性为一个持久化单元配置它。见代码清单 9.1。

代码清单9.1 在persistence.xml中自定义架构生成属性

路径: /model/src/main/resources/META-INF/persistence.xml

① 切换到多行提取器

② 定义应该何时执行脚本

```
<property name="hibernate.0hbm2ddl.import_files_sql_extractor"
          value="org.hibernate.tool.hbm2ddl.
          ↪MultipleLinesSqlCommandExtractor"/>

<property name="javax.persistence.schema-generation.create-source"
          value="script-then-metadata"/>

<property name="javax.persistence.schema-generation.drop-source"
          value="metadata-then-script"/>

<property name="javax.persistence.schema-generation.create-script-source"
          value="complexschemas/CreateScript.sql.txt"/>

<property name="javax.persistence.schema-generation.drop-script-source"
          value="complexschemas/DropScript.sql.txt"/>

<property name="javax.persistence.sql-load-script-source"
          value="complexschemas/LoadScript.sql.txt"/>
```

③ 创建架构的自定义 SQL 脚本

⑤ 加载脚本

④ 删除架构的自定义 SQL 脚本

① 默认情况下, Hibernate 会期望脚本中每行有一条 SQL 语句。这会切换到更方便的多行提取器。脚本中的 SQL 语句会以分号结尾。如果希望用另一种方式处理 SQL 脚本, 则可以编写你自己的 `org.hibernate.tool.hbm2ddl.ImportSqlCommandExtractor` 实现。

② 此属性会定义应该何时创建和删除脚本。自定义 SQL 脚本将包含 `CREATE DOMAIN` 语句, 必须在创建使用这些域的表之前执行它。有了这些设置, 架构生成器就会在读取 ORM 元数据(注解, XML 文件)和创建表之前首先运行该创建脚本。删除脚本会在 Hibernate 删除表之后执行, 给你提供清除所创建的任何内容的机会。其他选项是元数据(忽略自定义脚本源)和脚本(仅使用一个自定义脚本源; 忽略注解和 XML 文件中的 ORM 元数据)。

③ 这是用于架构创建的自定义 SQL 脚本的位置。其路径是(a)类路径上脚本资源的位置; (b)如 `file://` URL 的脚本位置; 或者, 如果(a)和(b)都不匹配, 则用(c)本地文件系统上的绝对或相对文件路径。本示例使用(a)。

④ 这是用于删除架构的自定义 SQL 脚本。

⑤ 这个加载脚本会在创建表之后运行。

我们已经提到过, DDL 通常是高度特定于供应商的。如果应用程序必须支持几种数据库方言, 那么可能就需要几组创建/删除/加载脚本来自定义用于每种数据库方言的架构。可以在 `persistence.xml` 文件中用几个持久化单元配置来解决这个问题。

或者, 在一个 `hbm.xml` 映射文件中, Hibernate 有其自己的用于架构自定义的专有配置。见代码清单 9.2。

代码清单9.2 使用Hibernate专有配置自定义架构生成

```
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <database-object>
        <create>
            CREATE ...
        </create>
        <drop>
            DROP ...
        </drop>
        <dialect-scope name="org.hibernate.dialect.H2Dialect"/>
        <dialect-scope name="org.hibernate.dialect.PostgreSQL82Dialect"/>
    </database-object>

</hibernate-mapping>
```

Hibernate 特性

将自定义 SQL 片段放入<create>和<drop>元素。Hibernate 会在为域模型类创建架构之后执行这些语句，会在创建表之后并且在删除架构的自动生成部分之前进行。这一行为无法变更，因此标准的 JPA 架构生成脚本设置提供了更多的灵活性。

<dialect-scope>元素将 SQL 语句限制为所配置数据库方言的一个特定集。如果没有任何<dialect-scope>元素，则将总是会应用该 SQL 语句。

Hibernate 也支持加载脚本：如果 Hibernate 在类路径的根节点发现一个名称为 import.sql 的文件，那么它就会在架构已经被创建之后执行该文件。或者，如果有几个导入文件，则可以在持久化单元配置中使用 hibernate.hbm2ddl.import_files 属性将它们命名为逗号分隔的列表。

最后，如果需要对所生成的架构具有更多编程控制，则可以实现 org.hibernate.mapping.AuxiliaryDatabaseObject 接口。Hibernate 捆绑了一个便利的实现，可以选择性地对其进行子类化和重写。见代码清单 9.3。

代码清单9.3 编程控制所生成的架构

```
package org.jpwh.model.complexschemas;

import org.hibernate.dialect.Dialect;
import org.hibernate.boot.model.relational.AbstractAuxiliaryDatabaseObject;

public class CustomSchema
    extends AbstractAuxiliaryDatabaseObject {
    public CustomSchema() {
        addDialectScope("org.hibernate.dialect.Oracle9Dialect");
    }

    @Override
    public String[] sqlCreateStrings(Dialect dialect) {
        return new String[]{"[CREATE statement]"};
    }
}
```



```
@Override
```

```
public String[] sqlDropStrings(Dialect dialect) {
    return new String[]{"[DROP statement]"};
}
```

可以程式添加方言作用域，甚至在 `sqlCreateString()` 和 `sqlDropString()` 方法中访问一些映射信息。必须在一个 `hbm.xml` 文件中启用此自定义类：

```
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">
    <database-object>
        <definition class="org.jpwh.model.complexschemas.CustomSchema"/>
        <dialect-scope name="org.hibernate.dialect.H2Dialect"/>
        <dialect-scope name="org.hibernate.dialect.PostgreSQL82Dialect"/>
    </database-object>
</hibernate-mapping>
```

附加的方言作用域是累加的；上一个示例应用了三种方言。

我们来编写一些自定义创建/删除/加载脚本并且实现所有优秀 DBA 都推荐的额外的架构完整性规则。首先，介绍一些关于完整性规则和 SQL 约束的背景信息。

9.1.2 SQL 约束

仅在应用程序代码中确保数据完整性的系统容易出现数据损坏，并且常常会随时间流逝降低数据库的质量。如果数据存储未强制规则，那么一个微小的未探测到的应用程序缺陷就会造成无法恢复的问题，比如错误数据或者数据丢失。

在与过程(或面向对象)应用程序代码中确保数据一致性不同，数据库管理系统允许像数据库架构那样使用声明实现完整性规则。声明式规则的好处在于，代码中的可能错误较少并且 DBMS 有机会优化数据访问。

在 SQL 数据库中，我们标识了四种规则：

- 域约束——域是数据库中的一种数据类型。因此，域约束定义了特定数据类型可以处理的可能值的范围。例如，INTEGER 数据类型可用于整数值。CHAR 数据类型可以保存字符串：例如，ASCII 中定义的所有字符或者其他一些字符编码。由于我们主要使用 DBMS 内置的数据类型，所以我们要依赖供应商定义的域约束。如果 SQL 数据库支持，那么就可以利用(通常受限的)支持为自定义域添加用于特定存在的数据类型的额外约束，或者创建用户定义的数据类型(UDT)。
- 列约束——限制一个列来保存特定域和类型的值可以创建一个列约束。例如，在架构中声明 EMAIL 列保存 VARCHAR 类型的值。或者，可以创建一个具有进一步约束的称为 EMAIL_ADDRESS 的新域，并用它替代 VARCHAR 应用到一个列。SQL 数据库中的特殊列约束是 NOT NULL。
- 表约束——应用到几列或几行的完整性规则就是一个表约束。典型的声明式表约束是 UNIQUE：要检查所有的行，不允许重复值(例如，每个用户都必须具有唯一

的电子邮件地址)。仅影响单个行但影响多个列的规则是“拍卖结束时间必须在拍卖开始时间之后。”

- 数据库约束——如果一个规则应用到多个表，那么它就具有数据库作用域。你应该已经很熟悉最常见的数据库约束了，它就是外键。这个规则会确保行之间的引用完整性，通常位于不同的表中，但也并不总是如此(自引用外键约束并不常见)。其他数据库约束涉及几个不常见的表：例如，只有在所引用商品的拍卖结束时间还未到时才能存储出价。

大多数 SQL 数据库管理系统都支持以上这些约束以及每种约束的大多数重要选项。除了类似 NOT NULL 和 UNIQUE 这样的简单关键字，通常还可以使用 CHECK 约束声明更复杂的规则，CHECK 约束应用任意 SQL 表达式。然而，完整性约束是 SQL 标准中较脆弱的区域之一，并且来自供应商的解决方案存在显著差异。

此外，进行拦截数据修改操作的数据库触发器可以使用非声明式和过程约束。然后触发器可以直接实现约束过程或者调用现有的存储过程。

在执行数据修改语句时，可以立即检查完整性约束，或者可以将该检查延迟到事务结束后。SQL 数据库中的违规响应通常是没有任何自定义可能的反射。外键是特殊的，因为通常可以决定对于所引用行的 ON DELETE 或 ON UPDATE 还会发生什么。

Hibernate 会在错误异常中传递数据库约束冲突；检查导致事务中的异常发生的原因，该原因是否位于 org.hibernate.exception.ConstraintViolationException 类型的异常链中的某个位置。此异常能够提供与错误有关的更多信息，比如失败的数据库约束的名称。

显示验证错误消息

这听起来似乎难以置信：数据库层将抛出一个包含所有详细信息的 ConstraintViolationException 异常，那么为何不将它显示给用户呢？之后用户可以在其屏幕上修改无效值，并且再次提交该表单直到数据通过有效性验证。遗憾的是，不能这么做，并且尝试实现这一策略的许多人都失败了。

首先，每个 DBMS 都有不同的错误消息，而 Hibernate 无法确保正确地解析错误。ConstraintViolationException 上提供的详细信息是最佳推测；它们通常都是错误的并且仅适用于开发人员记录消息日志。SQL 应该对此实现标准化吗？当然应该，但它没有这么做。

其次，应用程序不应该将无效数据传递到数据库来查看哪些遵循了约束、哪些没有遵循约束。DBMS 是最后一道防线，而非首个验证器。相反，要使用 Java 应用程序层中的 Bean 验证，并且以用户自己的语言向用户显示友好的验证错误消息。

我们详细介绍完整性约束的实现。

添加域和列约束

SQL 标准包括域，而遗憾的是域受到了相当限制，并且 DBMS 通常不支持域。如果系统支持 SQL 域，则可以使用它们将约束添加到数据类型。

在自定义 SQL 创建脚本中，基于 VARCHAR 数据类型定义一个 EMAIL_ADDRESS 域：

路径: /model/src/main/resources/complexschemas/CreateScript.sql.txt

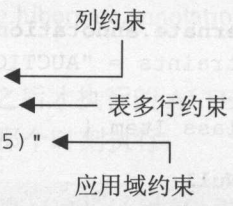
```
create domain if not exists
EMAIL_ADDRESS as varchar
check (position('@', value) > 1);
```

该附加的约束是对字符串中是否出现一个@符号的检查。SQL 中此域的(相对次要的)优势是将常见约束抽象成单个位置。在插入和修改数据时，总是会立即检查域约束。现在可以在映射中使用这个域，就像使用一个内置数据类型一样：

路径: /model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```
@Entity
public class User {

    @Column(
        nullable = false,
        unique = true,
        columnDefinition = "EMAIL_ADDRESS(255)"
    )
    protected String email;
    // ...
}
```



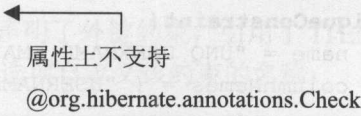
这个映射中呈现了几个约束。NOT NULL 约束是常见的；之前已经多次看到过它了。第二个是 UNIQUE 列约束；用户不能有重复的电子邮件地址。在编写本书时，很遗憾在 Hibernate 中没有办法自定义此单列唯一性约束的名称；在架构中它将得到自动生成的名称。最后，columnDefinition 引用了已经用自定义创建脚本添加过的域。此定义是一个 SQL 片段，它被直接导出到架构中，因此使用特定于数据库的 SQL 时要小心。

如果不希望首先创建域，则可以直接将 CHECK 关键字作为单列约束应用：

路径: /model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```
@Entity
public class User {

    @Column(columnDefinition =
        "varchar(15) not null unique" +
        " check (not substring(lower(USERNAME), 0, 5) = 'admin')"
    )
    protected String username;
    // ...
}
```



此约束将有效用户名值限制为最大 15 个字符长度，并且字符串不能以 admin 开头以避免混淆。可以调用任何 DBMS 支持的 SQL 函数；columnDefinition 总是会被传递到导出的架构中。

注意你有一个选择：创建并使用一个域或者添加一个单列约束具有相同的效果。域通

常更易于维护和避免重复。

在编写本书时，Hibernate 在单独的属性上还不支持其专有的注解 `@org.hibernate.annotations.Check`；可以将其用于表级别的约束。

Hibernate 特性

表级别约束

拍卖不能在其开始之前就结束。到目前为止 SQL 架构或 Java 域模型中并没有任何规则实现了此限制。需要一个单行表约束：

路径：/model/src/main/java/org/jpwh/model/complexschemas/custom/Item.java

```
@Entity
@org.hibernate.annotations.Check(
    constraints = "AUCTIONSTART < AUCTIONEND"
)
public class Item {

    @NotNull
    protected Date auctionStart;

    @NotNull
    protected Date auctionEnd;

    // ...
}
```

Hibernate 将表约束附加到了所生成的 CREATE TABLE 语句中，它可以包含任意 SQL 表达式。

可以使用更为复杂的表达式来实现多行表约束。在表达式中可能需要一个子查询来达到目的，DBMS 可能不支持该子查询。但有一些常见的多行表约束，比如 UNIQUE，可以直接添加到映射中。在上一节中，你已经看到过 `@Column(unique = true|false)` 选项。

如果唯一性约束覆盖到多个列，则要使用 `@Table` 注解的 `uniqueConstraints` 选项：

路径：/model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```
@Entity
@Table(
    name = "USERS",
    uniqueConstraints = {
        @UniqueConstraint(
            name = "UNQ_USERNAME_EMAIL",
            columnNames = { "USERNAME", "EMAIL" }
        )
    }
)
public class User {
    // ...
}
```

现在对于 USERS 表中的所有行来说，所有的 USERNAME 和 EMAIL 对都必须是唯一

的。如果不为该约束提供一个名称——这里使用了 `UNQ_USERNAME_EMAIL`——那么将使用一个自动生成且可能很难看的名称。

我们要探讨的最后几种约束是跨越多个表的数据库范围的规则。

数据库约束

在拍卖结束前，用户只能进行出价。数据库应该确保无效出价不能被存储，以便无论何时将一行插入到 `BID` 表中，都会对照拍卖结束时间检查该出价的 `CREATEDON` 时间戳。这类约束涉及两个表：`BID` 和 `ITEM`。

可以在任何 `SQL CHECK` 表达式的子查询中使用一个联结创建跨几个表的规则。相较于仅引用在其上声明了约束的表，可以查询(通常是为了特定信息片段的存在与否)另一个不同的表。问题在于，不能在 `Bid` 或 `Item` 类上使用 `@org.hibernate.annotations.Check` 注解。不知道 `Hibernate` 将首先创建哪个表。

因此，要将 `CHECK` 约束放到在所有的表已经创建之后才执行的 `ALTER TABLE` 语句中。加载脚本是一个合适的放置位置，因为它总是会在那个时刻执行：

路径：/model/src/main/resources/complexschemas/LoadScript.sql.txt

```
alter table BID
add constraint AUCTION_BID_TIME
check(
    CREATEDON <= (
        select i.AUCTIONEND from ITEM i where i.ID = ITEM_ID
    )
);
```

现在，`BID` 表中的一行是有效的，如果其 `CREATEDON` 值小于或等于所引用 `ITEM` 行的拍卖结束时间。

目前最常见的跨几个表的规则都是引用完整性规则。它们作为外键广为人知，它们是一个相关行的键值副本和一个确保所引用值存在的约束的组合。`Hibernate` 会为关联映射中的所有外键列自动创建外键约束。如果检查 `Hibernate` 生成的架构，就会注意到这些约束也具有自动生成的数据库标识符——其名称不易于阅读并且让调试更为困难。可以在所生成的架构中看到这类语句：

```
alter table BID add constraint FKCF AEEDB471BF59FF
foreign key (ITEM_ID) references ITEM
```

这条语句为 `BID` 表中的 `ITEM_ID` 列声明了外键约束，引用了 `ITEM` 表的主键列。可以在 `@JoinColumn` 映射中使用 `foreignKey` 选项自定义该约束的名称：

路径：/model/src/main/java/org/jpwh/model/complexschemas/custom/Bid.java

```
@Entity
public class Bid {

    @ManyToOne
    @JoinColumn(
```

```

        name = "ITEM_ID",
        nullable = false,
        foreignKey = @ForeignKey(name = "FK_ITEM_ID")
    )
    protected Item item;
    // ...
}

```

`@PrimaryKeyJoinColumn`、`@MapKeyJoinColumn`、`@JoinTable`、`@CollectionTable` 和 `@AssociationOverride` 映射中也支持 `foreignKey` 属性。

我们还没有介绍 `@ForeignKey` 注解具有的一些很少用到的选项：

- 可以编写你自己的 `foreignKeyDefinition`，像 `FOREIGN KEY([column]) REFERENCES [table]([column]) ON UPDATE [action]` 这样的 SQL 片段。Hibernate 将使用此 SQL 片段代替提供程序生成的片段，它可以位于 DBMS 所支持的 SQL 方言中。
- 如果希望完全禁用外键生成，则可以使用 `ConstraintMode` 设置，将值设置为 `NO_CONSTRAINT`。然后可以用一个 `ALTER TABLE` 语句编写外键约束，可能是在一个加载脚本中编写，就像我们介绍的一样。

正确命名约束不仅是一个好的实战，并且当必须读取异常消息时也有明显帮助。

这样就完成了我们对于数据库完整性规则的探讨。接下来，我们要查看出于性能原因可能希望在架构中包含的一些优化。

9.1.3 创建索引

在优化数据库应用程序的性能时，索引是一个关键功能。DBMS 中的查询优化器可以使用索引避免数据表的过度扫描。由于它们仅在数据库的物理实现中才具相关性，因此索引并非是 SQL 标准的一部分，并且 DDL 和可用的索引选项都是特定于产品的。不过，可以嵌入用于映射元数据中典型索引的最常见架构构件。

`CaveatEmptor` 中的许多查询都可能涉及 `User` 实体的 `username`。可以通过为这个属性的列创建一个索引来加速这些查询。索引的另一个候选是 `USERNAME` 和 `EMAIL` 列的组合，可以在查询中频繁使用它。可以在实体类上用 `@Table` 注解及其 `indexes` 属性声明单列索引或多列索引：

路径：/model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```

@Entity
@Table(
    name = "USERS",
    indexes = {
        @Index(
            name = "IDX_USERNAME",
            columnList = "USERNAME"
        ),
        @Index(
            name = "IDX_USERNAME_EMAIL",
            columnList = "USERNAME, EMAIL"
        )
    }
)

```



```
    )  
    }  
}  
public class User {  
    // ...  
}
```

如果不为该索引提供一个名称，则会使用生成的名称。

我们不推荐专门将索引添加到架构，因为这样就好像索引会有助于解决性能问题。如果希望学习有效的数据库优化技术，尤其是索引如何才能让你更熟悉对查询进行最佳性能执行规划方面的内容，那么可以参看 Dan Tow 所著的优秀书籍 *SQL Tuning* (Tow, 2003 年)。

自定义数据库架构通常仅在开发不具有已有数据的新系统时才可行。如果必须处理现有遗留架构，那么最常见的一个问题就是处理自然和组合键。

9.2 处理遗留键

我们在 4.2.3 节中提到过，我们认为自然主键会是一个不好的主意。自然键通常会造成功业务需求变更时难以修改数据模型。在极端情况下，它们甚至可能影响性能。遗憾的是，许多遗留架构都在很大程度上使用(自然)组合键；并且出于我们阻止使用组合键的原因，修改遗留架构以便使用非组合自然键或代理键可能会很困难。因此，JPA 支持自然的和组合的主键和外键。

9.2.1 映射一个自然主键

如果在遗留架构中遇到一个 USERS 表，那么可能 USERNAME 就是主键。在这种情况下，就没有 Hibernate 自动生成的代理标识符。相反，应用程序必须在保存 User 类的实例时指定标识符值：

```
路径: /examples/src/test/java/org/jpwh/test/complexschemas/ NaturalPrimaryKey.  
java
```

```
User user = new User("johndoe");  
em.persist(user);
```

此处，该用户的名称是 User 类唯一公共构造函数的一个参数：

```
路径: /model/src/main/java/org/jpwh/model/complexschemas/naturalprimarykey/  
User.java
```

```
@Entity  
@Table(name = "USERS")  
public class User {
```

```
    @Id  
    protected String username;
```

```

protected User() {
}

public User(String username) {
    this.username = username;
}

// ...
}

```

Hibernate 会在从数据库加载 User 时调用受保护的无参构造函数，然后直接指定 username 字段的值。当实例化一个 User 时，要用 username 调用该公共构造函数。如果不在 @Id 属性上声明一个标识符生成器，那么 Hibernate 就需要应用程序负责主键值的指定。

映射组合(自然)主键需要更多一些的工作量。

9.2.2 映射一个组合主键

假定 USERS 表的主键是 USERNAME 和 DEPARTMENTNR 这两列的组合。你要编写一个只声明键属性的单独组合标识符类，并且将这个类命名为 UserId：

路径: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/ embedded/
UserId.java

```

@Embeddable                                  ← ① @Embeddable, 可序列化的类
public class UserId implements Serializable {

    protected String username;                ← ② 自动 NOT NULL

    protected String departmentNr;

    protected UserId() {                      ← ③ 受保护的构造函数
    }

    public UserId(String username, String departmentNr) { ← ④ 公共构造函数
        this.username = username;
        this.departmentNr = departmentNr;
    }

    // ...

    ← ⑤ 重写 equals()和 hashCode()

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        UserId userId = (UserId) o;
        if (!departmentNr.equals(userId.departmentNr)) return false;
        if (!username.equals(userId.username)) return false;
        return true;
    }

    ← ⑤ 重写 equals()和 hashCode()

    @Override
    public int hashCode() {
        int result = username.hashCode();

```

```
        result = 31 * result + departmentNr.hashCode();
        return result;
    }
    // ...
}
```

- ❶ 这个类必须是@Embeddable 和 Serializable——JPA 中任何用作一个标识符类型的类型都必须是 Serializable。
- ❷ 不必将该组合键的属性标记为@NotNull；在被作为一个实体的主键嵌入时，它们的数据库列会自动变成 NOT NULL。
- ❸ JPA 规范需要一个用于可嵌入标识符类的公共无参构造函数。Hibernate 接受受保护的可见性。
- ❹ 该唯一的公共构造函数应该具有键值作为参数。
- ❺ 必须用数据库中组合键具有的相同语义来重写 equals()和 hashCode()方法。在这种情况下，这是 username 和 departmentNr 值的直接对比。

这些就是 UserId 类的重要部分。你大概也有一些获取方法来访问属性值。现在是用这个标识符类型作为@EmbeddedId 来映射 User 实体：

路径：/model/src/main/java/org/jpwh/model/complexschemas/compositekey/embedded/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @EmbeddedId
    protected UserId id;
    // ...
}
```

← 可选：@AttributeOverrides

就像为常规嵌入组件所做的那样，可以重写单独的属性及其映射的列，正如你在 5.2.3 节中所看到的一样。图 9-1 显示了该数据库架构。

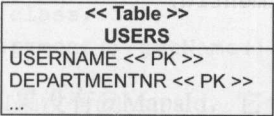


图 9-1 USERS 表具有一个组合主键

User 的任何公共构造函数都应该需要一个 UserId 的实例，以强制在保存该 User 之前提供一个值(当然，实体类必须具有另一个无参构造函数)：

路径: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyEmbeddedId.
java

```
UserId id = new UserId("johndoe", "123");
User user = new User(id);
em.persist(user);
```

这是加载一个 User 实例的方式:

路径: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyEmbeddedId.
java

```
UserId id = new UserId("johndoe", "123");
User user = em.find(User.class, id);
assertEquals(user.getId().getDepartmentNr(), "123");
```

接下来, 假定 DEPARTMENTNR 是引用 DEPARTMENT 表的一个外键, 并且希望在 Java 域模型中将这一关联表示为一个多对一关联。

9.2.3 组合主键中的外键

参看图 9-2 中的架构。

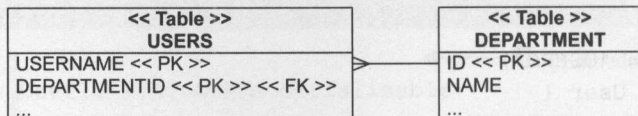


图 9-2 USERS 组合主键的一部分也是一个外键

第一个映射选项带有专门的注解 `@MapsId`, 是为此目的而设计的。首先将上一节中介绍的 `UserId` 嵌入标识符类中的 `departmentNr` 属性重命名为 `departmentId`:

路径: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/mapsid/
UserId.java

@Embeddable

```
public class UserId implements Serializable {
    protected String username;
    protected Long departmentId;
    // ...
}
```

该属性的类型现在是 Long, 而非 String。接着, 将具有 `@ManyToOne` 映射的 `department` 关联添加到 User 实体类:

路径: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/
mapsid/User.java

@Entity

```

@Table(name = "USERS")
public class User {

    @EmbeddedId
    protected UserId id;

    @ManyToOne
    @MapsId("departmentId")
    protected Department department;

    public User(UserId id) {
        this.id = id;
    }

    // ...
}

```

@MapsId 注解会告知 Hibernate 在保存一个 User 实例时忽略 UserId#departmentId 的值。Hibernate 会在将一个行保存到 USERS 表中时使用分配给 User#department 的 Department 的标识符：

路径: /examples/src/test/java/org/jpwh/test/complexschemas/ CompositeKeyMapsId.
java

```

Department department = new Department("Sales");
em.persist(department);

UserId id = new UserId("johndoe", null);    ← Null?
User user = new User(id);
user.setDepartment(department);             ← 必须的
em.persist(user);

```

Hibernate 会在保存时忽略设置为 UserId#departmentId 的任何值；此处它甚至被设置为 null。这意味着存储一个 User 时总是需要一个 Department 实例。JPA 将其称为一个派生的标识符映射。

当加载一个 User 时，只有 Department 的标识符是必要的：

路径: /examples/src/test/java/org/jpwh/test/complexschemas/ CompositeKeyMapsId.
java

```

UserId id = new UserId("johndoe", DEPARTMENT_ID);
User user = em.find(User.class, id);
assertEquals(user.getDepartment().getName(), "Sales");

```

我们不太喜欢此映射策略。如果没有 @MapsId，它会是一个更好的变体：

路径: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/
readonly/User.java

```

@Entity
@Table(name = "USERS")
public class User {

```

```

@EmbeddedId
protected UserId id;

@ManyToOne
@JoinColumn(
    name = "DEPARTMENTID",
    insertable = false, updatable = false
)
protected Department department;

public User(UserId id) {
    this.id = id;
}

// ...
}

```

← 默认为 DEPARTMENT_ID
← 让它只读

使用简单的 `insertable=false`、`updatable=false`，就能让 `User#department` 属性变为只读。那意味着只能通过调用 `someUser.getDepartment()` 查询数据，并且没有公共的 `setDepartment()` 方法。负责 `USERS` 表中 `DEPARTMENTID` 列的数据库更新的属性是 `UserId#departmentId`。

因此，现在必须在保存一个新 `User` 时设置部门的标识符：

路径: `/examples/src/test/java/org/jpwh/test/complexschemas/ CompositeKeyReadOnly.java`

```

Department department = new Department("Sales");
em.persist(department);

UserId id = new UserId("johndoe", department.getId());
User user = new User(id);
em.persist(user);

assertNull(user.getDepartment());

```

← 指定主键值
← 必须的
← 当心!

注意，`User#getDepartment()` 会返回 `null`，因为没有设置这个属性的值。Hibernate 只会在加载一个 `User` 时填充它：

路径: `/examples/src/test/java/org/jpwh/test/complexschemas/ CompositeKeyReadOnly.java`

```

UserId id = new UserId("johndoe", DEPARTMENT_ID);
User user = em.find(User.class, id);
assertEquals(user.getDepartment().getName(), "Sales");

```

许多开发人员都喜欢在一个构造函数中封装所有这些关注点：

路径: `/model/src/main/java/org/jpwh/model/complexschemas/compositekey/readonly/User.java`

```

@Entity
@Table(name = "USERS")
public class User {

```



```
public User(String username, Department department) {
    if (department.getId() == null)
        throw new IllegalStateException(
            "Department is transient: " + department
        );
    this.id = new UserId(username, department.getId());
    this.department = department;
}
// ...
}
```

这个防御型构造函数会强制 User 必须被实例化的方式以及正确设置所有标识符和属性值。

如果 USERS 表具有一个组合主键，那么引用该表的外键也必须是一个组合键。

9.2.4 引用组合主键的外键

例如，从 Item 到 User 的关联，即 seller，可能需要一个组合外键的映射。参看图 9-3 中的架构。

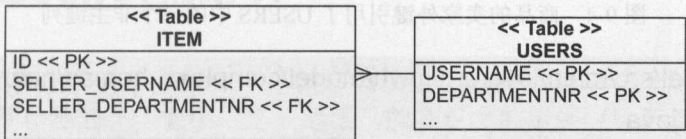


图 9-3 商品的卖家是用 ITEM 表中的组合外键表示的

Hibernate 可以在 Java 域模型中隐藏这一详情。这里是 Item#seller 属性的映射：

```
路径: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/
manytoone/Item.java
```

```
@Entity
public class Item {

    @NotNull
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name = "SELLER_USERNAME",
            referencedColumnName = "USERNAME"),
        @JoinColumn(name = "SELLER_DEPARTMENTNR",
            referencedColumnName = "DEPARTMENTNR")
    })
    protected User seller;

    // ...
}
```

你可能之前没有见过@JoinColumns 注解；它是此关联下面组合外键列的清单。要确保

提供了 `referencedColumnName` 属性，以便链接外键的源和目标。遗憾的是，如果忘记了，Hibernate 也不会提示，并且你可能最终会在所生成的架构中得到错误的列顺序。

在遗留架构中，外键有时不会引用一个主键。

9.2.5 引用非主键的外键

ITEM 表中 SELLER 列上的外键约束会通过要求在某个表的某行某列上提供相同卖家值来确保该商品的卖家存在。没有其他规则；目标列不需要一个主键约束或者一个唯一约束。目标表可以是任何一个表。值可以是卖家的一个数值标识符或者顾客的数字串；对于外键引用源和目标来说，只有类型必须是相同的。

当然，外键约束通常会引用一个或多个主键列。然而，有时遗留数据库会具有不遵循此简单规则的外键约束。有时一个外键约束会引用一个简单唯一列——自然的非主键。我们假设在 CaveatEmptor 中，如图 9-4 所示，需要处理 USER 表上被称为 CUSTOMERNR 的一个遗留自然键列：

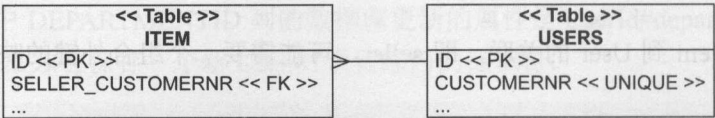


图 9-4 商品的卖家外键引用了 USERS 中的一个非主键列

路径: /model/src/main/java/org/jpwh/model/complexschemas/naturalforeignkey/
User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    @Column(unique = true)
    protected String customerNr;

    // ...
}
```

到目前为止，这没什么特殊的；你之前已经看到过这样的简单唯一属性映射。遗留的部分是 ITEM 表中的 SELLER_CUSTOMERNR 列，具有一个引用用户的 CUSTOMERNR 而非用户 ID 的外键约束：

路径: /model/src/main/java/org/jpwh/model/complexschemas/naturalforeignkey/
Item.java

```
@Entity
public class Item {
```

```
@NotNull
@ManyToOne
@JoinColumn(
    name = "SELLER_CUSTOMERNR",
    referencedColumnName = "CUSTOMERNR"
)
protected User seller;

// ...
}
```

要指定@JoinColumn 的 referencedColumnName 属性来声明此关系。Hibernate 现在就会知道，所引用的目标列是一个自然键，而非主键，并且会相应管理该外键关系。

如果目标自然键是一个组合键，则要像上一节中那样使用@JoinColumns 作为替代。幸运的是，清理这样的架构通常可以直接重构外键以引用主键——如果可以变更数据库而不会干扰其他共享该数据的应用程序的话。

这样就完成了我们关于在尝试映射遗留架构时可能必须处理的自然、组合以及外键相关问题的探讨。我们继续探讨另一个相关的特殊策略：将实体的基本或嵌入属性映射到辅助表。

9.3 将属性映射到辅助表

我们在 6.5 节的一个继承性映射中已经介绍过@SecondaryTable 注解了。它有助于将特定子类的属性取到单独的一个表中。这一通用功能有更多的用处——但要注意，一个恰当设计的系统应该具有简化的、比表更多的类。

假定在一个遗留架构中，没有在一个单独的表中保存一个用户的账单地址信息和其他用户详细信息。图 9-5 显示了此架构。该用户的家庭地址存储在 USERS 表的 STREET、ZIPCODE 和 CITY 列中。该用户的账单地址存储在 BILLING_ADDRESS 表中，该表的主键列是 USER_ID，它也是引用 USERS 表的 ID 主键的外键约束。

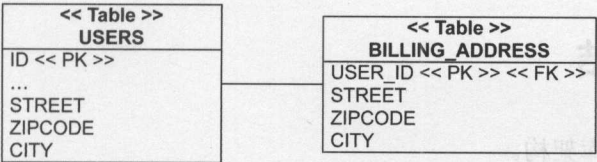


图 9-5 将账单地址数据取到辅助表中

要映射这个架构，需要为 User 实体声明辅助表，以及后续 Hibernate 应该如何用@SecondaryTable 联结它：

```
路径: /model/src/main/java/org/jpwh/model/complexschemas/secondarytable/
User.java
```

```
@Entity
@Table(name = "USERS")
@SecondaryTable(
```



```

    name = "BILLING_ADDRESS",
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "USER_ID")
)
public class User {

    protected Address homeAddress;

    @AttributeOverrides({
        @AttributeOverride(name = "street",
            column = @Column(table = "BILLING_ADDRESS",
                nullable = false)),
        @AttributeOverride(name = "zipcode",
            column = @Column(table = "BILLING_ADDRESS",
                length = 5,
                nullable = false)),
        @AttributeOverride(name = "city",
            column = @Column(table = "BILLING_ADDRESS",
                nullable = false))
    })
    protected Address billingAddress;

    // ...
}

```

User 类有两个嵌入类型的属性：homeAddress 和 billingAddress。第一个是一个常规嵌入映射，且 Address 类是 @Embeddable。

就像 5.2.3 节中一样，可以使用 @AttributeOverrides 注解来重写嵌入属性的映射。然后，@Column 会将单独的属性映射到 BILLING_ADDRESS 表，使用其 table 选项。记住，@AttributeOverride 为属性替换了所有的映射信息：如果进行了重写，则会忽略 Address 字段上的所有注解。因此，必须在 @Column 重写中再次指定为空性(nullability)和长度。

我们已经介绍了一个具有一个可嵌入属性的辅助表映射示例。当然，也可以将像 username 字符串这样的简单基本属性取到辅助表中。不过要牢记，读取和维护这些映射会是一个问题；应该仅用辅助表映射遗留的不可变架构。

9.4 本章小结

- 专注于数据库架构。
- 可以将额外的完整性规则添加到 Hibernate 生成的数据库架构。介绍了如何执行自定义创建、删除和加载 SQL 脚本。
- 探讨了使用 SQL 约束：域、列、表和数据库约束。
- 介绍了使用自定义 SQL 数据类型，以及检查、唯一性和外键约束。
- 探讨了一些在处理遗留架构、特别是处理键时必须解决的常见问题。
- 介绍了几种映射类型的知识：自然主键、组合主键、组合主键中的外键、映射到组合主键的外键，以及引用非主键的外键。
- 介绍了如何将一个实体的属性移动到辅助表中。

第Ⅲ部分

事务性数据处理

在第Ⅲ部分中，要用 Hibernate 和 Java 持久化加载与存储数据。将要采用编程接口、学习如何编写事务应用程序，以及 Hibernate 如何才能最有效地从数据库中加载数据。

从第 10 章开始，你将学习 JPA 应用程序中用于与实体实例交互的最重要的策略。你将看到实体实例的生命周期：它们如何变得持久化、分离以及移除。在第 10 章中，你会知道 JPA 中最重要的接口：EntityManager。接着，第 11 章会定义数据库和系统事务要点，以及如何使用 Hibernate 和 JPA 控制并发访问。你还会看到非事务数据访问。在第 12 章中，我们将介绍延迟和急加载、抓取计划、策略、配置文件以及用优化 SQL 执行进行封装。最后，第 13 章将介绍级联状态转换、侦听和拦截事件、用 Hibernate Envers 进行审核和版本控制，以及动态过滤数据。

在阅读完本部分之后，你将知道如何使用 Hibernate 和 Java 持久化编程接口以及如何有效加载、修改以及存储对象。你将理解事务如何工作以及为何会话处理能够为应用程序设计开辟新途径。你将做好准备对任何修改对象的场景进行优化以及应用最佳抓取和缓存策略来提高性能和可扩展性。

10.1 持久化生命周期

由于 JPA 是一个透明的持久化机制——类不需要它自己的持久化能力——可以编写

本章内容简介：

- 对象的生命周期和状态
- 处理 Java 持久化 API
- 处理分离状态

你现在理解了 Hibernate 和 ORM 如何解决对象/关系不匹配的静态方面。根据你目前所了解的知识，可以在 Java 类和 SQL 架构之间创建一个映射，以解决该结构性不匹配问题。作为正在解决的问题的提示，可以参阅 1.2 节。

一个有效的应用程序解决方案需要一些更多的东西：必须研究用于运行时数据管理的策略。这些策略对于应用程序的性能和正确行为至关重要。

在这一章中，我们将探讨实体实例的生命周期——实例如何变得持久化，以及如何停止考虑将其持久化——还有触发这些事务的方法调用和管理操作。JPA EntityManager 是访问数据的首选接口。

在介绍该 API 之前，首先探讨实体实例、其生命周期以及触发状态变更的事件。尽管有些资料可能过于正式，但对于持久化生命周期的切实理解是非常有必要的。

JPA 2 中主要的新特性

- 可以使用 `EntityManager#unwrap()` 得到该持久化管理器 API 特定于供应商的变体：例如 `org.hibernate.Session` API。使用 `EntityManagerFactory#unwrap()` 获得 `org.hibernate.SessionFactory`。
- 新的 `detach()` 操作提供了对持久化上下文的细粒度管理，以回收单独的实体实例。
- 从一个已有的 `EntityManager` 中，可以使用 `getEntityManagerFactory()` 获得用于创建持久化上下文的 `EntityManagerFactory`。
- 新的静态 `Persistence(Unit)Util` 帮助器方法会判定实体实例(或它的一个属性)是否被完整加载或者是否是一个未初始化的引用(Hibernate 代理或未加载的集合包)。

10.1 持久化生命周期

由于 JPA 是一个透明的持久化机制——类不清楚它们自己的持久化能力——可以编写

不清楚其所操作的数据是否表示持久化状态或仅存在于内存中的瞬时状态的应用程序逻辑。在调用其方法时，应用程序无须关心一个实例是否是持久化的。例如，可以调用 `Item#calculateTotalPrice()` 业务方法，而完全不必考虑持久化(例如在单元测试中)。

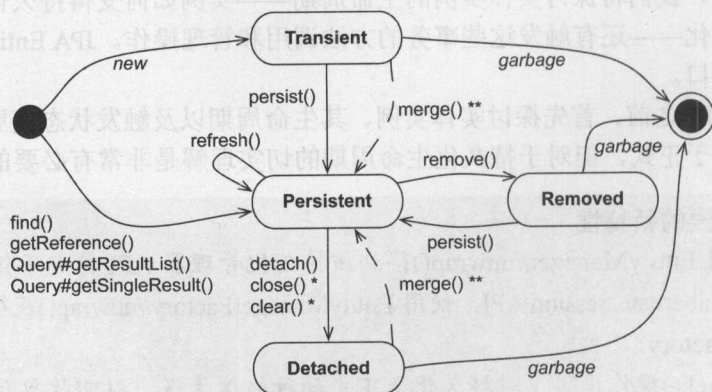
无论何时需要将内存中驻留的状态传递到数据库中(反之亦然)，任何具有持久化状态的应用程序都必须与持久化服务交互。换句话说，必须调用 Java 持久化接口来存储和加载数据。

在像这样与持久化机制交互时，应用程序必须从持久化方面关注实体实例的状态和生命周期。我们将其称为持久化生命周期：一个实体实例历经其生命的状态。我们还使用了工作单元这个术语：考虑一个(通常是原子的)组的一组(可能的)状态变更操作。另外一个问题是持久化服务提供的持久化上下文。将持久化上下文视作一个服务会让你想起在特定工作单元中对数据进行的所有修改和状态变更(这个说法有点简单，但它是一个好的开始)。

我们现在仔细研究所有这些术语：实体状态、持久化上下文以及托管作用域。你大概更习惯于思考必须管理的 SQL 语句以便将数据放入数据库以及从中取出；但成功使用 Java 持久化的一个关键因素是你对于状态管理的理解，因此请详细阅读这一节。

10.1.1 实体实例状态

不同的 ORM 解决方案会使用不同的专业术语并且会为持久化生命周期定义不同的状态以及状态迁移。此外，内部使用的状态可能与公开给客户端应用程序的那些状态不同。JPA 定义了四种状态，对客户代码隐藏了 Hibernate 内部实现的复杂性。图 10-1 显示了这些状态及其迁移。



* 影响持久化上下文中的所有实例

**合并会返回一个持久化实例，原始实例的状态不会变更

图 10-1 实体实例状态及其迁移

该状态图还包括对触发迁移的 `EntityManager`(和 `Query`)API 的方法调用。我们将在本章中探讨这个图；无论何时需要一份概述都可以参考它。我们来更为详尽地探讨这些状态和迁移。

状态迁移

使用新的 Java 操作符创建的实例是瞬时的，这意味着它们的状态会在不再引用它们时立即丢失和垃圾回收。例如，新的 `Item()` 会创建 `Item` 类的一个瞬时实例，就像新的 `Long()` 和新的 `BigDecimal()` 一样。Hibernate 未提供用于瞬时实例的任何回滚功能；如果修改一个瞬时 `Item` 的价格，那么就不能自动撤消该变更。

为了让一个实体实例从瞬时状态迁移成持久化状态，从而变成托管的，需要调用 `EntityManager#persist()` 方法或者创建来自一个已持久化实例的引用以及为所映射关联启用的状态级联。

持久化状态

在数据库中，持久化实体实例具有一个表示形式。它被存储在数据库中——或者在工作单元完成时被存储。它是一个具有数据库标识的实例，正如 4.2 节中所定义的；其数据库标识符被设置为数据库表示形式的主键值。

应用程序可能具有已创建的实例，然后通过调用 `EntityManager#persist()` 对它们进行持久化。在应用程序创建了对 JPA 提供程序已经管理的另一个持久化实例对象的引用时，可能就会有实例变成持久化。持久化实体实例可能是通过执行一个查询、通过一个标识符查找，或者通过开始于另一个持久化实例的对象图导航来从数据库中检索到的实例。

持久化实例总是与持久化上下文相关联的。稍后你将看到更多与此有关的内容。

移除状态

可以用几种方式从数据库中删除一个持久化实体实例：例如，可以使用 `EntityManager#remove()` 来移除它。如果从已映射的集合中移除对持久化实体实例的引用，那么也可以借助启用的孤儿删除来删除该持久化实体实例。

之后实体实例就处于移除状态了：提供程序会在工作单元结束时删除它。应该在完成对它的处理后丢弃应用程序中可能持有的对它的所有引用——例如，在已经为用户呈现了移除确认界面后就应该这样做。

分离状态

为理解分离的实体实例，要考虑加载一个实例。通过其(已知的)标识符调用 `EntityManager#find()` 来检索一个实体实例。然后结束工作单元并且关闭持久化上下文。应用程序仍然具有一个句柄——对加载的实例的引用。它现在就处于分离状态了，并且数据变得过时了。可以丢弃该引用并且让垃圾回收器回收内存。或者，可以继续处理处于分离状态的数据，然后调用 `merge()` 方法在新的工作单元中保存修改。我们将在本章稍后专门的一节中再次探讨分离和合并。

你现在应该有了对实体实例状态及其迁移的一个基本理解了。我们的下一个主题是持久化上下文：所有 Java 持久化提供程序的一个必要服务。

10.1.2 持久化上下文

在一个 Java 持久化应用程序中，`EntityManager` 具有一个持久化上下文。要在调用

`EntityManagerFactory#createEntityManager()` 时创建一个持久化上下文。当调用 `EntityManager#close()` 时, 会关闭该上下文。在 JPA 专业术语中, 这是一个应用程序托管的持久化上下文; 应用程序会定义该持久化上下文的作用域, 以界定工作单元。

该持久化上下文会监控并管理处于持久化状态的所有实体。该持久化上下文是 JPA 提供程序大部分功能的核心。

该持久化上下文允许持久化引擎执行自动的脏检查, 以检测应用程序修改了哪些实体实例。然后提供程序会与数据库同步受持久化上下文监控的实例状态, 要么自动要么按需。通常来说, 当个工作单元完成时, 提供程序会通过执行 SQL INSERT、UPDATE 和 DELETE 语句(所有的数据修改语言[DML])来将驻留在内存中的状态传递到数据库。这一刷新过程也可能发生在其他时刻。例如, Hibernate 可以在查询执行之前同步数据库。这样就能确保查询知晓早前工作单元期间所做的变更。

持久化上下文会充当一级缓存; 它会记住在特定工作单元中处理过的所有实体实例。例如, 如果要求 Hibernate 使用主键值(通过标识符的查找)加载一个实体实例, Hibernate 就会首先检查持久化上下文中的当前工作单元。如果 Hibernate 在持久化上下文中找到了该实体实例, 则不会发生数据库访问——这是应用程序的反复读取。对相同持久化上下文的连续 `em.find(Item.class, ITEM_ID)` 调用会产生相同的结果。

此缓存还会影响任意查询的结果, 比如使用 `javax.persistence.Query` API 执行的查询。Hibernate 会读取查询的 SQL 结果集并且将它转换成实体实例。这一过程首先会尝试通过标识符查找解析持久化上下文中的每一个实体实例。只有在当前持久化上下文中无法找到具有相同标识符值的实例时, Hibernate 才会从结果集中读取其余的数据。如果实体实例已经位于持久化上下文中, 那么 Hibernate 会忽略该结果集中任何可能的较新数据, 这是由于数据库级的提交读事务隔离造成的。

该持久化上下文缓存总是开启的——它无法被关闭。它会确保达成以下目标:

- 持久化层不容易在对象图形中的环形引用的情况下出现堆栈溢出。
- 在工作单元结束时不能有相同数据库行的冲突表示形式。提供程序可以将所有对实体实例的变更安全地写到数据库。
- 同样, 特定持久化上下文中进行的变更总是会立即对工作单元及其持久化上下文内执行的所有其他代码可见。JPA 会确保可重复的实体-实例读取。

持久化上下文会提供对象标识的保障范围; 在单个持久化上下文的范围中, 只有一个实例表示特定数据库行。思考一下引用 `entityA == entityB` 的比较。只有当两者在堆上都引用了相同 Java 实例时, 该比较结果才为 `true`。现在, 思考 `entityA.getId().equals(entityB.getId())` 这一对比。如果两者都具有相同的数据库标识符值, 那么结果就是 `true`。在一个持久化上下文内, Hibernate 会确保两个比较产生相同的结果。这样就解决了 1.2.3 节中我们介绍过的基础 O/R 不匹配问题。

作用于进程的标识是否更好?

对于一个典型的 Web 或企业应用程序来说, 作用于持久化上下文的标识会更好。作用于进程的标识, 其中只有一个内存实例表示整个处理进程(JVM)中的行, 它将在缓存利用方面提供一些潜在的优势。不过, 在一个广泛使用多线程的应用程序中, 在全局标识映射

中总是同步对持久化实例的共享访问的代价会高得难以承受。在每个持久化上下文中让每个线程处理一个数据的独立副本会更简单且更可扩展。

一开始,由持久化上下文提供的实体实例和服务的生命周期会难以理解。参看与脏检查、缓存以及受保障的标识作用域如何实际运行有关的一些代码示例。为此,要使用持久化管理器 API。

10.2 EntityManager 接口

所有的透明持久化工具都包含一个持久化管理器 API。这个持久化管理器通常会提供用于基础 CRUD(创建、读取、更新、删除)操作、查询执行以及控制持久化上下文的服务。在 Java 持久化应用程序中,你与之交互的主要接口是 `EntityManager`,以创建工作单元。

10.2.1 规范的工作单元

在 Java SE 和一些 EE 架构中(例如,如果只有普通的服务器小应用),通过调用 `EntityManagerFactory#createEntityManager()` 将得到一个 `EntityManager`。应用程序代码会共享 `EntityManagerFactory`,这表示一个持久化单元或一个逻辑数据库。大多数应用程序都只有一个共享的 `EntityManagerFactory`。

要在单个线程中为单个工作单元使用 `EntityManager`,并且创建它的代价很低。代码清单 10.1 显示了一个工作单元的规范、典型形式。

代码清单 10.1 一个典型的工作单元

路径: `/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java`

```
EntityManager em = null;
UserTransaction tx = TM.getUserTransaction();
try {
    tx.begin();
    em = JPA.createEntityManager();    ← 应用程序托管的
    // ...

    tx.commit();                      ← 同步/刷新持久化上下文
} catch (Exception ex) {
    // Transaction rollback, exception handling
    // ...
} finally {
    if (em != null && em.isOpen())
        em.close();                  ← 创建它,就要关闭它!
}
```

(`TM` 类是本书示例代码附带的一个方便的类。此处它简化了 JNDI 中标准 `UserTransaction` API 的查找。该 `JPA` 类提供了对共享的 `EntityManagerFactory` 的便利访问)。

`tx.begin()` 和 `tx.commit()` 之间的所有一切都发生在一个事务中。目前,要牢记,像由 Hibernate 执行的 SQL 语句这样的事务作用域中的所有数据库操作要么完全成功,要么完

全失败。现在不要太担心事务代码；你将在下一章中读到更多与并发控制有关的内容。我们将再次查看相同的示例，这次要专注于事务和异常处理代码。不过，不要在你的代码中编写空的 catch 子句，否则你必须回滚事务并且处理异常。

创建一个 EntityManager 就启动了其持久化上下文。Hibernate 只在必要时才访问数据库；EntityManager 只有在 SQL 语句必须被执行时才会从池中获得一个 JDBC Connection。可以在不访问数据库的情况下创建和关闭一个 EntityManager。Hibernate 会在你查找或查询数据以及将由持久化上下文检测到的变更刷新到数据库时执行 SQL 语句。Hibernate 会在创建一个 EntityManager 以及等待事务提交时参与处理中的系统事务。当(JTA 引擎)通知 Hibernate 有提交时，它就会执行持久化上下文的脏检查并且与数据库同步。你还可以在事务期间的任何时候通过调用 EntityManager#flush()来手动强制脏检查同步。

要通过选择何时对该 EntityManager 执行 close()操作来决定持久化上下文的作用域。有时你必须关闭该持久化上下文，因此应该总是将 close()调用放在 finally 代码块。

持久化上下文应该开启多长时间？我们假设对于以下示例你正在编写一个服务端，并且每个客户端请求将被多线程环境中的一个持久化上下文和系统事务处理。如果熟悉服务器小应用，可以想象一下代码清单 10.1 中的代码嵌入到一个服务器小应用的 service()方法中的情况。在此工作单元内部，你要访问 EntityManager 来加载和存储数据。

10.2.2 使数据持久化

我们来创建一个实体的新实例并且将它从瞬时状态转换成持久化状态：

路径：/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = new Item();
item.setName("Some Item");
em.persist(item);
Long ITEM_ID = item.getId();
```

← Item#name 是 NOT NULL

← 已经被分配

可以在图 10-2 中看到相同的工作单元以及 Item 实例如何变更状态。

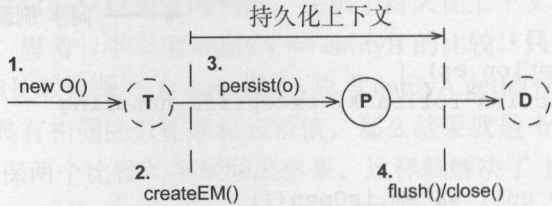


图 10-2 在工作单元中让一个实例持久化

像通常一样实例化一个新的瞬时 Item。当然，也可以在创建 EntityManager 之前实例化它。调用 persist()会让该 Item 的瞬时实例持久化。它现在由当前持久化上下文托管并且与之关联。

为了在数据库中存储 Item 实例，Hibernate 必须执行 SQL INSERT 语句。当此工作单

元的事务提交时, Hibernate 会刷新该持久化上下文, 而 INSERT 会在那时发生。Hibernate 甚至可以在 JDBC 级别将 INSERT 与其他语句一起批量处理。当调用 `persist()` 时, 仅会分配 Item 的标识符值。或者, 如果标识符生成器并非在每次插入时执行, 则 INSERT 语句将在调用 `persist()` 时立即执行。可以回顾一下 4.2.5 节。

使用标识符检测实体状态

有时需要知道一个实体实例是瞬时的、持久化的还是分离的。如果 `EntityManager#contains(e)` 返回 `true`, 则实体实例就处于持久化状态。如果 `PersistenceUnitUtil#getIdentifier(e)` 返回 `null`, 则它就处于瞬时状态。如果它不是持久化的, 并且 `PersistenceUnitUtil#getIdentifier(e)` 返回该实体标识符属性的值, 则它就处于分离状态。可以从 `EntityManagerFactory` 中得到 `PersistenceUnitUtil`。

有两个问题值得注意。首先要注意, 在刷新持久化上下文之前, 标识符值可能并未指定和可用。第二, 如果标识符属性是一个基元(long, 并非 Long), 则 Hibernate(不同于其他一些 JPA 提供程序)不会从 `PersistenceUnitUtil#getIdentifier()` 处返回 `null`。

最好(但并非必须)在使用持久化上下文管理 Item 实例之前完全初始化它。SQL INSERT 语句包含调用 `persist()` 时由实例持有的值。如果不在让其持久化之前设置 Item 的 name, 就可能违背 NOT NULL 约束。可以在调用 `persist()` 之后修改该 Item, 并且你的修改将被额外的 SQL UPDATE 语句传递到数据库。

如果在刷新失败时执行了一个 INSERT 或 UPDATE 语句, 那么 Hibernate 就会在数据库级别触发此事务中对持久化实例所做的变更的回滚。但 Hibernate 不会回滚内存中对持久化实例所做的变更。如果在 `persist()` 之后修改了 `Item#name`, 那么提交失败将不会回滚到旧的名称。这是合理的, 因为事务的失败通常都是不可恢复的, 并且必须立即丢弃该失败的持久化上下文和 `EntityManager`。我们将在下一章中探讨异常处理。

接下来, 你要加载并且修改所存储的数据。

10.2.3 检索和修改持久化数据

可以使用 `EntityManager` 从数据库中检索持久化实例。对于接下来的示例, 我们假设你已经在某处保存了上一节中所存储的 Item 的标识符值, 并且你现在正在一个新的工作单元中根据标识符查找相同的实例:

路径: `/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java`

```
Item item = em.find(Item.class, ITEM_ID);
```

```
if (item != null)
```

```
    item.setName("New Name");
```

← 修改

← 如果还没有在持久化上下文中, 则访问数据库

图 10-3 用图形的形式显示了这一转换。

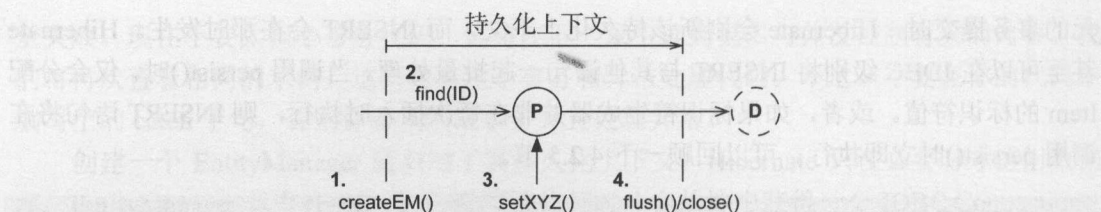


图 10-3 在工作单元中让一个实例持久化

你无须转换 `find()` 操作的返回值；它是一个通用方法，并且其返回类型被设置为第一个参数的意外产物。检索的实体实例处于持久化状态，并且你现在可以在工作单元中修改它。

如果无法找到具有指定标识符值的持久化实例，则 `find()` 将返回 `null`。如果没有找到持久化上下文缓存中的指定实体类型和标识符，那么 `find()` 操作就会总是访问数据库。实体实例总是在加载期间初始化的。可以预期稍后在分离状态中让其所有的值可用：例如，在你关闭持久化上下文后呈现一个界面时(如果启用了 Hibernate 的可选二级缓存，则它可能不会访问数据库；我们将在 20.2 节中探讨此共享缓存)。

可以修改 `Item` 实例，持久化上下文将检测这些变更并且自动在数据库中记录它们。当 Hibernate 在提交期间刷新持久化上下文时，它会执行必要的 SQL DML 语句来将变更同步到数据库。Hibernate 会尽可能迟地在事务结束时将状态变更传递到数据库。DML 语句通常会在数据库中创建锁，这些锁会保留到事务完成时，这样 Hibernate 就可以尽可能短地在数据库中保持锁的周期。

Hibernate 会使用一个 SQL UPDATE 将新的 `Item#name` 写入数据库。默认情况下，Hibernate 会在 SQL UPDATE 语句中包含所映射 ITEM 表的所有列，以便将未变更的列更新为其旧的值。因此，Hibernate 可以在启动而非运行时生成这些基本 SQL 语句。如果希望在 SQL 语句中仅包含修改过(或者用于 INSERT 的非可空)的列，就能启用 4.3.2 节中探讨过的动态 SQL 生成。

在从数据库中加载了 `Item` 时，Hibernate 会通过将该 `Item` 与它之前使用的快照副本做比较来检测变更过的 `name`。如果 `Item` 不同于快照，则有必要使用 UPDATE。持久化上下文中的这个快照会消耗内存。使用快照的脏检查也会消耗时间，因为 Hibernate 必须在刷新期间使用其快照对比持久化上下文中的所有实例。

你可能希望使用一个扩展点来自定义 Hibernate 检测脏状态的方式。在 `persistence.xml` 配置文件中将属性 `hibernate.entity_dirtiness_strategy` 设置为实现 `org.hibernate.CustomEntityDirtinessStrategy` 的类的名称。更多信息可以参阅这个接口的 Javadoc。`org.hibernate.Interceptor` 是用于自定义脏检查的另一个扩展点，通过实现其 `findDirty()` 方法来达成。可以在 13.2.2 节中找到一个示例拦截器。

我们之前提到过，持久化上下文启用了可重复的实体实例读取并且提供了对象标识的保障：

路径：`/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java`

```
Item itemA = em.find(Item.class, ITEM_ID);
Item itemB = em.find(Item.class, ITEM_ID);
```

← 可重复的读取

```
assertTrue(itemA == itemB);
assertTrue(itemA.equals(itemB));
assertTrue(itemA.getId().equals(itemB.getId()));
```

第一个 `find()` 操作会访问数据库并且使用一个 `SELECT` 语句检索该 `Item` 实例。第二个 `find()` 是在持久化上下文中完成的，并且会返回相同缓存的 `Item` 实例。

有时需要一个实体实例，但不希望访问数据库。

10.2.4 得到一个引用

如果不希望在加载一个实体实例时访问数据库，因为你不确定是否需要一个完全初始化的实例，那么可以告知 `EntityManager` 尝试提取一个空的占位符——一个代理：

路径：/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = em.getReference(Item.class, ITEM_ID); ← ❶getReference()

PersistenceUnitUtil persistenceUtil = ← ❷帮助器方法
    JPA.getEntityManagerFactory().getPersistenceUnitUtil();
assertFalse(persistenceUtil.isLoaded(item));

// assertEquals(item.getName(), "Some Item"); ← ❸初始化代理
// Hibernate.initialize(item); ← ❹加载代理数据

tx.commit();
em.close();

assertEquals(item.getName(), "Some Item"); ← ❺分离状态中的 item
```

- ❶ 如果持久化上下文已经包含具有指定标识符的一个 `Item`，那么该 `Item` 实例就是通过 `getReference()` 而不是访问数据库来返回的。此外，如果当前没有托管具有该标识符的持久化实例，那么 `Hibernate` 就会生成一个空占位符：一个代理。这意味着 `getReference()` 不会访问数据库，并且它不会返回 `null`，不同于 `find()`。
- ❷ `JPA` 提供了 `isLoaded()` 这样的 `PersistenceUnitUtil` 帮助器方法来检测你是否正在使用一个未初始化的代理。
- ❸ 只要在该代理上调用任意方法，比如 `Item#getName()`，就会执行 `SELECT` 来完全初始化占位符。此规则的例外是映射的数据库标识符获取方法，比如 `getId()`。代理可能看起来像真实存在的，但它仅仅是一个持有它所表示的实体实例的标识符值的占位符而已。如果初始化代理时数据库记录不再存在，那么会抛出一个 `EntityNotFoundException` 异常。注意该异常会在调用 `Item#getName()` 时抛出。
- ❹ `Hibernate` 有一个方便的静态 `initialize()` 方法，它会加载代理的数据。
- ❺ 在关闭持久化上下文之后，`item` 就处于分离状态。如果不在持久化上下文仍旧开启时初始化代理，那么当访问该代理时就会得到一个 `LazyInitializationException` 异常。无法在持久化上下文被关闭时按需加载数据。解决方案很简单：在关闭持久化上下文之前加载数据。

在第 12 章中，我们将探讨更多与代理、延迟加载和按需提取有关的内容。
如果希望从数据库中移除一个实体实例的状态，那么就必须让其变成瞬时的。

10.2.5 让数据变成瞬时的

要让一个实体实例变成瞬时的并且删除其数据库表示，可以调用 EntityManager 上的 remove()方法：

路径: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = em.find(Item.class, ITEM_ID);  ← ❶调用 find()或 getReference()
//Item item = em.getReference(Item.class, ITEM_ID);

em.remove(item);  ← ❷将实例排队以便删除

assertFalse(em.contains(item));  ← ❸检查实体状态

// em.persist(item);  ← ❹取消删除

assertNull(item.getId());  ← ❺提交事务

tx.commit();
em.close();
```

启用了 hibernate.use_identifier_rollback;
现在看起来像是瞬时实例了

- ❶ 如果调用 find(), 则 Hibernate 会执行一个 SELECT 来加载 Item。如果调用 getReference(), 则 Hibernate 会尝试避免 SELECT 并返回一个代理。
- ❷ 在工作单元完成时调用 remove() 会将实体实例排队以便删除；它现在就处于移除状态。如果 remove() 是在一个代理上调用的，则 Hibernate 会执行一个 SELECT 来加载数据。在生命周期转换期间必须完全初始化一个实体实例。可以使用生命周期回调方法或者启用一个实体侦听器(参阅 13.2 节)，并且实例必须通过这些拦截器来完成其完整的生命周期。
- ❸ 移除状态中的实体不再处于持久化状态。可以用 contains() 操作检查这一点。
- ❹ 可以让移除的实例再次持久化，取消删除。
- ❺ 当事务提交时，Hibernate 会与数据库同步状态迁移并且执行 SQL DELETE。JVM 垃圾回收器会检测 item 是否不再被引用，并且最终删除数据的最后痕迹。

图 10-4 显示了相同的处理过程。

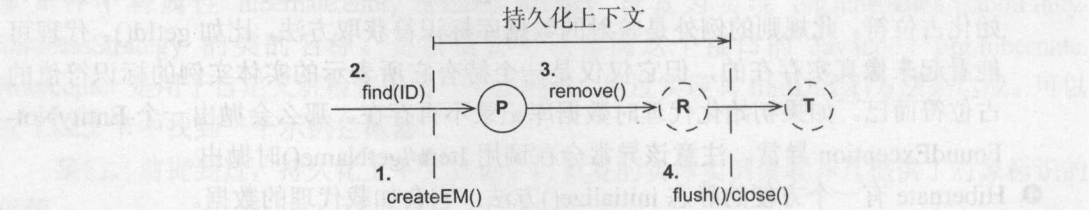


图 10-4 移除工作单元中的实例

默认情况下，Hibernate 不会修改已移除实体实例的标识符值。这意味着 item.getId() 方法仍旧会返回当前已过期的标识符值。有时进一步使用“已删除的”数据是有用的：例

如, 如果用户决定撤消操作, 那么可能希望再次保存已移除的 Item。正如示例中所示的, 可以在已移除的实例上调用 `persist()` 在刷新持久化上下文之前取消删除。或者, 如果在 `persistence.xml` 中将属性 `hibernate.use_identifier_rollback` 设置为 `true`, 则 Hibernate 将在实体实例移除后重新设置标识符值。在上一个代码示例中, 标识符值被重新设置为默认值 `null` (它是一个 `Long`)。现在该 Item 与瞬时状态中的相同, 并且可以在新的持久化上下文中再次保存它。

Java 持久化还提供了大量在应用程序中转换成不具有生命周期拦截器的直接 SQL `DELETE` 语句的操作。我们将在 20.1 节中探讨这些操作。

我们假设你从数据库中加载一个实体实例并处理该数据。出于某些原因, 你知道另一个应用程序或者可能是你的应用程序的另一个线程已经更新了数据库中的基础行。接下来, 我们看看如何刷新内存中驻留的数据。

10.2.6 刷新数据

以下示例揭示了一个持久化实体实例的刷新:

路径: `/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java`

```
Item item = em.find(Item.class, ITEM_ID);
item.setName("Some Name");

// Someone updates this row in the database

String oldName = item.getName();
em.refresh(item);
assertNotEquals(item.getName(), oldName);
```

在加载该实体实例之后, 会意识到其他人修改了数据库中的数据(并不重要)。调用 `refresh()` 会引发 Hibernate 执行 `SELECT` 来读取和收集整个结果集, 重写已经对应用程序内存中持久化实例进行的变更。如果数据库行不再存在(某人删除了它), 那么 Hibernate 会在 `refresh()` 上抛出一个 `EntityNotFoundException` 异常。

大多数应用程序都不必手动刷新内存中的状态; 并发修改通常是在事务提交时完成的。刷新的最佳用例是使用一个扩展的持久化上下文, 它可能跨几个请求/响应周期和/或系统事务。当等待用户对开放的持久化上下文输入时, 数据会变得过时, 并且可选择刷新可能需要取决于用户和系统之间会话和对话的持续时间。如果用户取消对话, 那么刷新对于撤销会话期间内存中所做的修改就是有用的。我们在 18.3 节中将介绍更多与会话中刷新有关的内容。

另一个频繁使用的操作是实体实例的复制。

10.2.7 复制数据

复制是有用的, 比如当需要从一个数据库检索数据并且在另一个数据库中存储它的时候。复制会使用一个持久化上下文中加载的分离实例并且让它们在另一个持久化上下文中

持久化。通常要从两个不同的 `EntityManagerFactory` 配置中打开这些上下文，启用两个逻辑数据库。必须在这两个配置中映射该实体。

`replicate()`操作仅在 `Hibernate Session API` 上可用。这里是从一个数据库加载 `Item` 实例并将它复制到另一个数据库中的示例：

路径：/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
tx.begin();

EntityManageremA = getDatabaseA().createEntityManager();
Item item = emA.find(Item.class, ITEM_ID);

EntityManageremB = getDatabaseB().createEntityManager();
emB.unwrap(Session.class)
    .replicate(item, org.hibernate.ReplicationMode.LATEST_VERSION);

tx.commit();
emA.close();
emB.close();
```

对这两个数据库的连接可以参与到相同的系统事务中。

`ReplicationMode` 会控制复制过程的详情：

- **IGNORE**——当数据库中有一个具有相同标识符的已有数据库行时，就忽略实例。
- **OVERWRITE**——重写数据库中具有相同标识符的任何已有数据库行。
- **EXCEPTION**——如果目标数据库中有一个具有相同标识符的已有数据库行时，就抛出一个异常。
- **LATEST_VERSION**——如果其版本比指定实体实例的版本旧，则重写数据库中的行，或者忽略该实例。需要启用具有实体版本控制的乐观并发控制。

当使输入到不同数据库中的数据保持一致时，你可能就需要复制。一个示例是产品升级：如果应用程序的新版本需要一个新的数据库(架构)，那么你可能就希望一次性移植并且复制已有数据。

持久化上下文会为你做许多事情：自动脏检查、确保对象标识的作用域等。了解它的一些管理详情，并且有时会影响后台的运行情况是相当重要的。

10.2.8 在持久化上下文中缓存

持久化上下文是持久化实例的缓存。持久化状态中的每个实体实例都与持久化上下文相关。

许多忽略此简单情况的 `Hibernate` 用户都会碰到 `OutOfMemoryException` 异常。这通常出现在你在工作单元中加载数千个实体实例却从不打算修改它们时。`Hibernate` 仍旧必须在持久化上下文缓存中创建每个实例的快照，这会导致内存耗尽(显然，如果修改数千行，则应该执行批数据操作——我们将在 20.1 节中回过头来介绍这种工作单元)。

持久化上下文缓存不会自动收缩。将持久化上下文的大小保持为必要的最低限度。通常，上下文中的许多持久化实例都是偶然出现的——例如，因为你只需要一些项却查询了许多。非常大的图形会具有严重的性能影响并且需要用于状态快照的大量内存。检查你的

查询仅返回需要的数据，并且考虑以下几种控制 Hibernate 的缓存行为的方式。

可以调用 `EntityManager#detach(i)` 手动从持久化上下文中收回持久化实例。可以调用 `EntityManager#clear()` 来分离所有的持久化实体实例，为你留下一个空的持久化上下文。

原生的 Session API 具有一些对你可能有用的额外操作。可以将整个持久化上下文设置为只读架构。这会禁用状态快照和脏检查，而且 Hibernate 不会将修改写入数据库：

路径: `/examples/src/test/java/org/jpwh/test/fetching/ReadOnly.java`

```
em.unwrap(Session.class).setDefaultReadOnly(true);

Item item = em.find(Item.class, ITEM_ID);
item.setName("New Name");

em.flush(); ←—— 没有 UPDATE
```

可以为单个实体实例禁用脏检查：

路径: `/examples/src/test/java/org/jpwh/test/fetching/ReadOnly.java`

```
Item item = em.find(Item.class, ITEM_ID);

em.unwrap(Session.class).setReadOnly(item, true);
item.setName("New Name");

em.flush(); ←—— 没有 UPDATE
```

使用 `org.hibernate.Query` 接口的查询可以返回只读结果，Hibernate 不会对其进行修改检查：

路径: `/examples/src/test/java/org/jpwh/test/fetching/ReadOnly.java`

```
org.hibernate.Query query = em.unwrap(Session.class)
    .createQuery("select i from Item i");

query.setReadOnly(true).list();

List<Item> result = query.list();

for (Item item : result)
    item.setName("New Name");

em.flush(); ←—— 没有 UPDATE
```

归功于查询访问，还可以为使用 JPA 标准 `javax.persistence.Query` 接口获得的实例禁用脏检查：

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.READ_ONLY,
        true
    );
```


要当心只读实体实例：仍然可以删除它们并对集合进行修改会很棘手！如果使用这些设置处理映射的集合，那就需要阅读 **Hibernate** 使用手册提供的一长列特殊用例。将在第 14 章中看到更多的查询示例。

到目前为止，持久化上下文的刷新和同步都已经在事务提交时自动发生了。在一些情况下，需要对同步进程有更多的控制。

10.2.9 刷新持久化上下文

默认情况下，无论何时提交连接的事务，**Hibernate** 都会刷新 **EntityManager** 的持久化上下文并且与数据库同步变更。除了上一节中的一些示例，所有之前的代码示例都使用了该策略。**JPA** 允许实现在其他时候同步持久化上下文，如果需要这样做的话。

作为一个 **JPA** 实现，**Hibernate** 会在以下情况进行同步：

- 当提交连接的 **JTA** 系统事务时
- 在执行一个查询之前——我们不打算用 `find()` 查找，而是使用 `javax.persistence.Query` 或类似的 **Hibernate** API 进行查询
- 当应用程序显式调用 `flush()` 时

可以用 **EntityManager** 的 `FlushModeType` 设置控制此行为：

路径：/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();           ❶加载 Item 实例

Item item = em.find(Item.class, ITEM_ID);
item.setName("New Name");                               ❷变更实例名称

em.setFlushMode(FlushModeType.COMMIT);                 ❸在查询前禁用刷新

assertEquals(
    em.createQuery("select i.name from Item i where i.id = :id")
        .setParameter("id", ITEM_ID).getSingleResult(),
    "Original Name"                                     ❹得到实例名称
);

tx.commit();      ❺刷新!
em.close();
```

这里，加载了一个 **Item** 实例❶并修改了其名称❷。然后查询了该数据库，检索该项的名称❸。通常 **Hibernate** 会识别出该数据已经在内存中修改了并在查询之前与数据库同步这些修改。这是 `FlushModeType.AUTO` 的行为，将 **EntityManager** 与事务连接时的默认行为。使用 `FlushModeType.COMMIT`，就是在查询之前禁用刷新，可以看到由该查询返回的不同于存在于内存中的数据。然后同步仅会在事务提交时发生。

可以在事务处于处理过程中的任何时候通过调用 `EntityManager#flush()` 来强制进行脏检查和与数据库同步。

这样就结束了我们对于瞬时、持久化和移除实体状态，以及 **EntityManager** API 的基本使用的探讨。控制这些状态迁移和 API 方法是必要的；每一个 **JPA** 应用程序都是内置在这

些操作中的。

接下来，我们查看分离的实体状态。我们已经提到了你将在实体实例不再与持久化上下文相关时看到的一些问题，比如禁用的延迟初始化。我们用一些示例来探究分离的状态，这样就知道在处理持久化上下文之外的数据时该有何期望。

10.3 处理分离的状态

如果一个引用脱离了受保障的标识的作用域，则我们就称其为对一个分离的实体实例的引用。当持久化上下文被关闭时，它就不再提供一个标识映射的服务。当处理分离的实体实例时，就会碰到混淆的问题，因此要确保你理解如何处理分离实例的标识。

10.3.1 分离实例的标识

如果在相同的持久化上下文中使用相同的数据库标识符值查找数据，那么其结果就是在 JVM 堆上有对相同内存中实例的两个引用。思考一下代码清单 10.2 显示的两个工作单元。

代码清单10.2 Java持久化中受保障的对象标识作用域

路径: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
tx.begin(); // ①创建持久化上下文
em = JPA.createEntityManager(); // ②加载实体实例

Item a = em.find(Item.class, ITEM_ID); // ③a 和 b 具有相同 Java 标识
Item b = em.find(Item.class, ITEM_ID);
assertTrue(a == b); // ④a 和 b 是相等的
assertEquals(a.getId(), b.getId()); // ⑤a 和 b 具有相同数据库标识

tx.commit(); // ⑥提交事务
em.close(); // ⑦关闭持久化上下文

tx.begin();
em = JPA.createEntityManager();

Item c = em.find(Item.class, ITEM_ID); // ⑧a 和 c 不相同
assertTrue(a != c); // ⑨a 和 c 不相等
assertFalse(a.equals(c));
assertEquals(a.getId(), c.getId()); // ⑩标识验证仍旧为 true

tx.commit();
em.close();
```

在第一个工作单元的 `begin()` 处①，首先要创建一个持久化上下文②并且加载相同的实体实例。由于引用 `a` 和 `b` 是从相同的持久化上下文中获得的，因此它们具有相同的 Java 标识③。它们是相等的④，因为默认情况下 `equals()` 依赖于 Java 标识对比。它们显然具有相同的数据库标识⑤。在持久化状态中，它们引用相同的 `Item` 实例，受该工作单元的持久化

上下文管理。这个示例的第一个部分是通过提交事务⑥和关闭持久化上下文⑦来完成的。在关闭第一个持久化上下文时，引用 *a* 和 *b* 都处于分离状态。你正在处理存在于一个受保障对象标识作用域之外的实例。

可以看到，在不同持久化上下文中加载的 *a* 和 *c* 是不相同的⑧。使用 *a.equals(c)* 进行的相等性测试也是 *false*⑨。对数据库标识的测试仍旧返回 *true*⑩。如果将实体实例当作与在分离状态中相等，那么此行为就会导致问题。例如，在第二个工作单元结束之后，思考一下代码的以下扩展：

```
em.close();

Set<Item> allItems = new HashSet<>();
allItems.add(a);
allItems.add(b);
allItems.add(c);
assertEquals(allItems.size(), 2);
```

那看起来是错误
并且随意的

这个示例将所有三个引用添加到了一个 *Set*。它们都是对分离实例的引用。现在，如果检查该集合的大小——元素的数量——你期望得到什么结果？

Set 不允许重复元素。重复元素是通过 *Set* 检测到的；无论何时添加一个引用，就会对已经位于该集合中的所有其他元素自动调用 *Item#equals()* 方法。如果 *equals()* 为任何已经位于该集合中的元素返回 *true*，则不会出现相加。

默认情况下，所有 Java 类都会继承 *java.lang.Object* 的 *equals()* 方法。此实现使用双等号(=)比较来检查两个引用是否涉及 Java 堆上的相同内存中实例。

你可能会猜测该集合中元素的数量是 2。毕竟 *a* 和 *b* 是对相同内存中实例的引用；它们已经在相同的持久化上下文中加载了。你从另一个持久化上下文中获得了引用 *c*；它涉及堆上的另一个实例。你有对两个实例的三个引用，因为你已经看到了加载了该数据的代码。在一个真实的应用程序中，你可能不知道相较于 *c*，*a* 和 *b* 是在不同的上下文中加载的。此外，你显然会期望该集合正好具有一个元素，因为 *a*、*b* 和 *c* 代表相同的数据库行，即相同的 *Item*。

无论何时处理分离状态中的实例以及测试它们是否相等(通常在基于哈希的集合中)，都需要为所映射的实体类提供 *equals()* 和 *hashCode()* 方法的实现。这是一个重要的问题：如果不处理分离状态中的实体实例，就不需要操作，并且 *java.lang.Object* 的默认 *equals()* 实现就够用了。要依赖持久化上下文中的 *Hibernate* 的受保障的对象标识作用域。即便你处理分离实例：如果永远不检查它们是否相等，就不要将其放在一个 *Set* 中或将其用作 *Map* 中的键。如果所做的就是在界面上呈现一个分离的 *Item*，就不要将它与其他东西做比较。

许多刚开始接触 JPA 的开发人员都认为他们总是必须为所有的实体类提供一个自定义的相等性例程，但情况并非如此。在 18.3 节中，我们将介绍一个用扩展的持久化上下文策略设计的应用程序。此策略还会扩展受保障对象标识的作用域，以跨越整个会话和几个系统事务。注意你仍然需要不对比在两个会话中获得的分离实例的原则！

我们假设你希望使用分离实例并且必须使用你自己的方法对其进行相等性测试。

10.3.2 实现相等性方法

可以用几种方式来实现 `equals()` 和 `hashCode()` 方法。要牢记当重写 `equals()` 时，还总是需要重写 `hashCode()` 以便两个方法保持一致。如果两个实例是相等的，则它们必须具有相同的哈希值。

一个看似明智的方法是实现 `equals()` 来仅对比数据库标识符属性，它通常是一个代理主键值。基本上，如果两个 `Item` 实例具有由 `getId()` 返回的相同标识符，则它们必须是相同的。如果 `getId()` 返回 `null`，那么它必须是还未保存的瞬时 `Item`。

遗憾的是，此解决方案有一个大问题：标识符值在实例变成持久化之后才会被 Hibernate 指定。如果在被保存之前将一个瞬时实例添加到 `Set`，那么当保存它时，其哈希值就会在它被 `Set` 包含时发生变更。这与 `java.util.Set` 的契约相反，破坏了该集合。尤其是，这个问题会让级联持久化状态对于基于集的映射关联无用。我们强烈反对数据库标识符的相等性。

为了实现我们推荐的解决方案，需要理解业务键的概念。业务键是一个属性，或者属性的一些组合，即对于每个具有相同数据库标识的实例是唯一的。实质上，如果不使用一个代理主键作为替代，那么它就是你要使用的自然键。不同于一个自然主键，业务键永远不会变更并非是一个绝对需求——只要它很少变更，就足够了。

我们认为，实质上每一个实体类都应该具有一个业务键，即便它包括该类的所有属性（这对于一些不可变类来说是合适的）。如果用户正在屏幕上查看一组商品，他们如何在商品 A、B 和 C 之间进行区分？相同的属性，或者属性的组合就是你的业务键。业务键就是用户视作唯一标识某特定记录的东西，而代理键是应用程序和数据库系统依赖的东西。业务键的一个或多个属性在你的数据库架构中最可能被约束为 `UNIQUE`。

我们为 `User` 实体类编写自定义相等性方法；这对比 `Item` 实例要容易。对于 `User` 类，`username` 是一个很好的候选业务键。它总是必要的，它对于数据库约束是唯一的，并且它很少变更，即便有也不多。见代码清单 10.3。

代码清单 10.3 User 相等性的自定义实现

```
@Entity
@Table(name = "USERS",
        uniqueConstraints =
            @UniqueConstraint(columnNames = "USERNAME"))
public class User {

    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (!(other instanceof User)) return false; ← User instanceof
        User that = (User) other;
        return
            this.getUsername().equals(that.getUsername()); ← Use getter
    }
```

```

@Override
public int hashCode() {
    return getUsername().hashCode();
}

// ...
}

```

你可能已经注意到，`equals()`方法代码总是会通过 `getter` 方法访问“其他”引用的属性。这非常重要，因为作为 `other` 传递的引用可能是一个 `Hibernate` 代理，而非保持持久化状态的实际实例。你不能直接访问 `User` 代理的 `username` 字段。要初始化该代理来获得属性值，需要使用一个 `getter` 方法来访问它。这是 `Hibernate` 并非完全透明的一个点，但无论如何，使用 `getter` 方法替代直接的实例变量访问是一种好的做法。

使用 `instanceof` 而不是通过对比 `getClass()` 的值来检查其他引用的类型。此外，其他引用可能是一个代理，它是 `User` 的一个运行时生成的子类，因此 `this` 和 `other` 可能不是完全相同的类型，而是一个有效的超/子类型。可以在 12.1.1 节中看到更多与代理有关的内容。

你现在可以安全对比持久化状态中的 `User` 引用：

```

tx.begin();
em = JPA.createEntityManager();

User a = em.find(User.class, USER_ID);
User b = em.find(User.class, USER_ID);
assertTrue(a == b);
assertTrue(a.equals(b));
assertEquals(a.getId(), b.getId());

tx.commit();
em.close();

```

当然，此外，如果比较对持久化和分离状态中实例的引用，就会得到正确的行为：

```

tx.begin();
em = JPA.createEntityManager();

User c = em.find(User.class, USER_ID);
assertFalse(a == c);           ← 当然，仍旧是 false
assertTrue(a.equals(c));       ← 现在是 true
assertEquals(a.getId(), c.getId());

tx.commit();
em.close();

Set<User> allUsers = new HashSet();
allUsers.add(a);
allUsers.add(b);
allUsers.add(c);
assertEquals(allUsers.size(), 1); ← 正确！

```

对于其他一些实体来说，业务键可能更为复杂，由一组属性构成。这里是一些提示，

它们会帮助你识别域模型类中的业务键：

- 思考一下，应用程序的用户会在他们必须标识一个(现实环境中的)对象时参考哪些属性。如果一个元素和另一个元素被显示在屏幕上，那么用户要如何在两者之间进行区分？这大概就是你正在寻求的业务键。
- 每一个不可变属性都可能是业务键的合适候选。如果可变的属性很少被更新或者如果可以控制它们被更新的情况，那么它们可能也是合适的候选——例如，在更新时确保实例不在一个 Set 中。
- 每个具有 UNIQUE 数据库约束的属性都是业务键的合适候选。记住，业务键的精确度必须足够合适，以避免重叠。
- 任何基于日期或时间的属性，比如记录的创建时间戳，通常都是业务键的一个合适组件，但 `System.currentTimeMillis()` 的精确度取决于虚拟机和操作系统。我们推荐的安全缓冲是 50 毫秒，这在基于时间的属性是业务键的单个属性时可能并非足够精确。
- 可以将数据库标识符用作业务键的一部分。这看起来与我们之前的描述相矛盾，但我们不是在讨论指定实体的数据库标识符值。可以使用相关联实体实例的数据库标识符。例如，Bid 类的候选业务键就是它与出价金额一起匹配的 Item 的标识符。甚至可以使用数据库架构中代表这个组合业务键的唯一约束。可以使用相关联 Item 的标识符值，因为它不会在 Bid 的生命周期期间变更——Bid 构造函数会需要一个已经持久化的 Item。

如果遵循我们的建议，就不会在为所有的业务类查找合适业务键时碰到太多困难。如果碰到了困难的情况，可以尝试在不考虑 Hibernate 的情况下解决它。毕竟，它纯粹是一个面向对象问题。注意，重写子类上的 `equals()` 并且在对比中包含另一个属性几乎都不会是正确的。在这种情况下，满足 Object 标识以及相等性同时具有均称性和可传递性的相等性需求会有些棘手；以及更为重要的是，业务键可能并不对应数据库中任何良好定义的候选自然键(子类属性可能被映射到不同的表)。要了解更多与自定义相等性比较有关的信息，可以参阅 Joshua Bloch 所著的 *Effective Java*, 2nd Edition(Bloch, 2008)，一本适用于所有 Java 编程人员的必读书籍。

User 类现在做好了用于分离状态的准备；可以在不同持久化上下文中加载的实例安全地放到一个 Set 中。接下来，我们查看一些涉及分离状态的示例，并且你将看到此概念的一些好处。

有时你可能希望手动从持久化上下文中分离一个实体实例。

10.3.3 分离实体实例

不必等待持久化上下文关闭。可以手动回收实体实例：

路径：/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
User user = em.find(User.class, USER_ID);
```

```
em.detach(user);
```



```
assertFalse(em.contains(user));
```

这个示例还演示了 `EntityManager#contains()` 操作，它会在指定实例在这个持久化上下文中处于托管持久化状态时返回 `true`。

现在可以处理分离状态中的 `user` 引用。许多应用程序都会在持久化上下文关闭之后仅读取和呈现数据。

在持久化上下文被关闭之后修改加载的 `user` 对数据库中它的持久化表示形式没有影响。不过，JPA 允许你将任何变更合并回新的持久化上下文中的数据库内。

10.3.4 合并实体实例

我们假设你已经在前一个持久化上下文中检索了 `User` 实例，并且你现在希望修改它以及保存这些修改：

路径：/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
detachedUser.setUsername("johndoe");
```

```
tx.begin();
```

```
em = JPA.createEntityManager();
```

在 `merging.mergedUser` 处于持久化状态之后丢弃 `detachedUser` 引用

```
User mergedUser = em.merge(detachedUser);
```

```
mergedUser.setUsername("doejohn");
```

```
tx.commit();
```

```
em.close();
```

数据库中的 UPDATE

思考图 10-5 中这一过程的图形化表示。它不像看起来那么难。

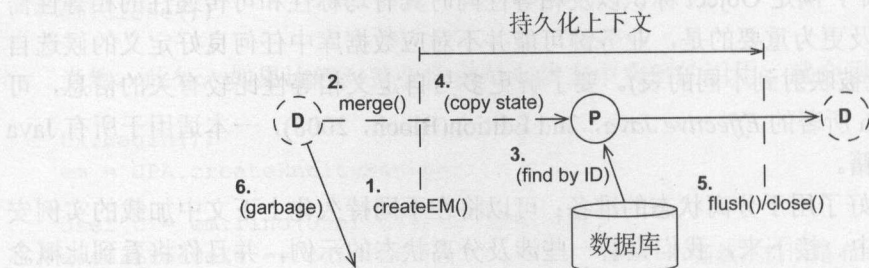


图 10-5 在工作单元中让一个实例持久化

目标是记录分离 `User` 的新 `username`。首先，当调用 `merge()` 时，Hibernate 会检查持久化上下文中的持久化实例是否具有作为你正在合并的分离实例的相同数据库标识符。

在这个示例中，该持久化上下文为空；没有从数据库中加载任何东西。因此 Hibernate 会从数据库中加载具有这个标识符的实例。然后，`merge()` 会复制该分离的实体实例到这个加载的持久化实例上。换句话说，你已经在分离的 `User` 上设置的新 `username` 也会设置在持久化合并的 `User` 上，它是 `merge()` 为你返回的。

现在丢弃对过时和过期分离状态的旧引用；`detachedUser` 不再表示当前状态。可以继续修改返回的 `mergedUser`；Hibernate 将在提交期间刷新该持久化上下文的时候执行单个

UPDATE。

如果该持久化上下文中没有具有相同标识符的持久化实例，并且数据库中根据标识符的查找失败，那么 Hibernate 就会实例化一个新的 User。然后 Hibernate 会将你的分离实例复制到这个新实例上，Hibernate 会在你将持久化上下文同步到数据库时将它插入到数据库中。

如果赋予 `merge()` 的实例并非是分离的，而是瞬时的(它不具有一个标识符值)，那么 Hibernate 就会实例化一个新的 User，将该瞬时 User 的值复制到它之上，然后让其持久化并且将其返回给你。简单来说，`merge()` 操作可以处理分离和瞬时的实体实例。Hibernate 总是会返回作为持久化实例的结果。

基于分离和合并的应用程序架构可能不会调用 `persist()` 操作。可以合并新的和分离的实体实例来存储数据。重要的区别是所返回的当前状态以及如何处理应用程序代码中引用的此切换。必须丢弃 `detachedUser` 并且从现在起引用当前 `mergedUser`。应用程序中仍旧保持在 `detachedUser` 上的其他每一个组件都必须切换成 `mergedUser`。

我能复位一个分离的实例吗？

Hibernate Session API 有一个用于复位的称为 `saveOrUpdate()` 的方法。它会接受一个瞬时实例或者一个分离实例，并且不会返回任何内容。该操作之后，指定实例将处于持久化状态，因此你不必切换引用。如果指定实例是瞬时的，那么 Hibernate 会执行一个 INSERT，或者它是分离的，则 Hibernate 会执行 UPDATE。我们建议你转而借助合并，因为它是标准的并且因此更易于集成其他框架。此外，相较于 UPDATE，如果未修改分离数据，则合并可能仅会触发一个 SELECT。如果想知道该 Session API 的 `saveOrUpdateCopy()` 方法的功能，那么可以将其看作与 EntityManager 上的 `merge()` 相同。

如果希望删除一个分离实例，就必须首先合并它。然后调用由 `merge()` 返回的该持久化实例上的 `remove()`。

我们将在第 18 章中再次查看分离状态和合并，并且使用此策略实现用户和系统之间的一个更为复杂的会话。

10.4 本章小结

- 我们探讨过最重要的策略以及用于在 JPA 应用程序中与实体实例交互的一些可选策略。
- 你了解了与实体实例的生命周期和它们如何变得持久化、分离和移除有关的内容。
- JPA 中最重要的接口是 EntityManager。
- 在大多数应用程序中，数据并非单独存储和加载。Hibernate 通常是在多用户应用程序中集成的，并且数据库是在许多线程中被并发访问的。

事务和并发

11

第 11 章

本章内容简介:

- 定义数据库和系统事务要素
- 使用 Hibernate 和 JPA 控制并发访问
- 使用非事务性数据访问

在本章中我们终于要谈论事务了：如何在应用程序中创建和控制并发的的工作单元。工作单元是操作的一个原子分组。事务允许你设置工作单元边界并且帮助你将多个工作单元隔离开。在多用户应用程序中，还可以并发处理这些工作单元。

要处理并发性，我们首先要专注于最低层的工作单元：数据库和系统事务。你将学习用于事务分界的 API 以及如何在 Java 代码中定义工作单元。我们将探讨如何保持隔离以及使用悲观和乐观策略控制并发访问。

最后，我们要查看一些特殊的用例和 JPA 功能，基于没有显式事务的数据库访问。首先我们要介绍一些背景信息。

JPA 2 中主要的新功能

- 悲观锁有一些新的锁模式和异常。
- 可以在 Query 上设置一个锁模式，悲观锁或乐观锁。
- 可以在调用 EntityManager#find()、refresh() 或 lock() 时设置一个锁模式。悲观锁模式的锁超时提示也是标准的。
- 当抛出新的 QueryTimeoutException 或 LockTimeoutException 异常时，事务不必回滚。
- 持久化上下文现在可以处于具有禁用的自动刷新的非同步模式中。这允许你在联结一个事务前对修改进行排队，并且从事务中解耦 EntityManager 的使用。

11.1 事务的要素

应用程序功能需要一次性完成几件事情。例如，在拍卖结束时，CaveatEmptor 应用程序必须执行三个不同的任务：

- (1) 为拍卖商品找出中标出价(最高的金额)。

(2) 对商品卖家收取拍卖的费用。

(3) 通知卖家和中标者。

如果由于外部信用卡系统的故障你不能结算拍卖费用的话会发生什么？该业务需求可能会规定，要么所有列出的操作必须都成功，要么必须都失败。如果是这样的话，就可以将这些步骤共同称为一个事务或工作单元。即便只有单个步骤失败，整个工作单元也必须失败。

11.1.1 ACID 属性

ACID 表示原子性、一致性、隔离性、持久性。原子性指的是一个事务中的所有操作都作为一个原子单元执行。此外，事务允许多个用户在不损坏数据一致性的情况下并发地使用相同数据(与数据库完整性规则一致)。一个特定的事务不应对其他并发运行的事务可见；它们应该隔离运行。即便系统在事务已经成功完成之后失败，一个事务中做出的变更也应该是持久的。

另外，你会希望确保事务的准确性。例如，业务规则规定应用程序对卖家收费一次，而非两次。这是一个合理的假设，但可能无法用数据库约束来表述它。因此，一个事务的准确性就是应用程序的职责了，而一致性是数据库的职责。所有这些事务属性一起定义了 ACID 标准。

11.1.2 数据库和系统事务

我们还提到过系统和数据库事务。再次思考一下最后一个示例：在工作单元结束一次拍卖期间，我们可能会在数据库系统中标记中标价。然后，在相同的工作单元中，我们要告知一个外部系统结算卖家的信用卡。这是一个跨几个(子)系统的事务，在可能的几个资源上具有配套的从属事务，比如一个数据库连接和一个外部结算处理器。

数据库事务必须短，因为乐观的事务会消耗数据库资源并且可能会避免并发访问，这是由于数据上的独占锁造成的。单个数据库事务通常只涉及单批次数据库操作。

为了执行系统事务中所有的数据库操作，必须设置该工作单元的边界。必须启动该事务并且在某些时候，提交变更。如果发生了一个错误(要么在执行数据库操作时，要么在提交事务时)，你必须回滚变更以便让数据处于一致性状态。此处理过程定义了事务分界，并且依据你使用的技术，还涉及某种级别的人工干预。一般而言，开始和结束一个事务的事务边界可以在应用程序代码中编程式或声明式地设置。

11.1.3 使用 JTA 的编程式事务

在 Java SE 环境中，可以调用 JDBC API 来标记事务边界。要在 JDBC Connection 上使用 `setAutoCommit(false)` 来开始一个事务，并且通过调用 `commit()` 来结束它。在事务处于处理过程的任何时候，你都可以强制使用 `rollback()` 来立即进行回滚。

在操作几个系统数据的应用程序中，某特定工作单元涉及访问多个事务性资源。在这种情况下，你无法单独用 JDBC 实现原子性。需要能够在一个系统事务中处理几个资

源的事务管理器。JTA 标准化了系统事务管理和分布式事务，这样就不必过于担心较低层的详情。JTA 中的主要 API 是 `UserTransaction` 接口，它具有 `begin()` 和 `commit()` 系统事务的方法。

其他的事务分界 API

JTA 提供了底层资源的事务系统的一个好的抽象，以及分布式系统事务的额外优势。许多开发人员仍旧认为只能得到具有运行在 Java EE 应用程序服务器中的组件的 JTA。如今，像 Bitronix(本书的示例代码中用到了它)和 Atomikos 这样的高质量独立 JTA 提供程序是可用的，并且易于安装在任何 Java 环境中。可以将这些解决方案视作 JTA 启用的数据库连接池。

无论何时，在可能的情况下都应该使用 JTA，并且避免专有的事务 API，比如 `org.hibernate.Transaction` 或者非常受限的 `javax.persistence.EntityTransaction`。这些 API 是在 EJB 运行时容器之外 JTA 无法轻易使用时创建的。

在 10.2.1 节中，我们许诺过要再次从异常处理的角度查看事务。代码清单 11.1 就是该代码，这一次完成了回滚和异常处理。

代码清单 11.1 具有事务边界的典型工作单元

路径: `/examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java`

```
EntityManager em = null;
UserTransaction tx = TM.getUserTransaction();
try {
    tx.begin();
    em = JPA.createEntityManager(); ← 应用程序托管的
    // ...
    tx.commit(); ← 同步/刷新持久化上下文
} catch (Exception ex) { ← 事务回滚; 异常处理
    try {
        if (tx.getStatus() == Status.STATUS_ACTIVE
            || tx.getStatus() == Status.STATUS_MARKED_ROLLBACK)
            tx.rollback();
    } catch (Exception rbEx) {
        System.err.println("Rollback of transaction failed, trace follows!");
        rbEx.printStackTrace(System.err);
    }
    throw new RuntimeException(ex);
} finally {
    if (em != null && em.isOpen())
        em.close(); ← 创建了它，就要关闭它
}
```

此代码片段最复杂的一点看起来就是异常处理；稍后我们将探讨这一部分。首先，必须理解事务管理和 `EntityManager` 如何共同工作。

`EntityManager` 是延迟的；我们在上一章中提到过，在必须执行 SQL 语句之前，它不

会消耗任何数据库连接。这同样适用于 JTA：当没有访问任何事务资源时，开启和提交一个空事务的成本很低。例如，可以在一个服务器上为每个客户端请求执行此空工作单元，而无需消耗任何资源或保留任何数据库锁。

当创建 `EntityManager` 时，它会在当前执行线程内查找一个运行中的 JTA 系统事务。如果 `EntityManager` 找到了一个运行中的事务，就会通过侦听事务事件来联结该事务。这意味着如果想要联结它们的话，应该总是在相同线程上调用 `UserTransaction#begin()` 和 `EntityManagerFactory#createEntityManager()`。默认情况下，如第 10 章所述，Hibernate 会在事务提交时自动刷新持久化上下文。

如果创建 `EntityManager` 时无法在相同线程中找到一个开启了的事务，那么它就会处于特殊的非同步模式。在这个模式中，JPA 不会自动刷新持久化上下文。我们将在本章稍后更多地探讨这一行为；在你设计更为复杂的会话时，它就是 JPA 的一个便利功能。

常见问题解答：回滚只读事务是否更快？

如果事务中的代码读取数据而不修改它，那么应该回滚该事务而不是提交它吗？这样是否更快？显然，有些开发人员发现这在某些特殊环境下会更快，并且此观点已经传遍了社区。我们用更流行的数据库系统验证了这一点并且没有找到差异。我们也没有发现任何显示出性能差异的实数来源。没有原因表明数据库系统应该具有次优的清理实现——所以为何不在内部使用最快的事务清理算法呢。

如果提交失败，则应总是提交你的事务并且进行回滚。话虽如此，但 SQL 标准包含了语句 `SET TRANSACTION READ ONLY`。我们建议你首先调研你的数据库是否支持这一点并且查看可能的性能优势是什么，如果有的话。

当一个事务运行的时间过长时，事务管理器就会停止该事务。记住，在一个忙碌的 OLTP 系统中，你希望尽可能短地保持数据库事务。默认的超时设置取决于 JTA 提供程序——例如，Bitronix 默认设置为 60 秒。可以在开始事务之前使用 `UserTransaction#setTransactionTimeout()` 进行选择性地重写。

我们仍旧需要探讨前一个代码片段的异常处理。

11.1.4 处理异常

如果提交期间任何 `EntityManager` 调用或持久化上下文刷新抛出了一个异常，那么就必须检查系统事务的当前状态。当发生异常时，Hibernate 会标记该事务以便回滚。这意味着这个事务的唯一可能输出就是撤消其所有变更。因为你开启了该事务，所以检查 `STATUS_MARKED_ROLLBACK` 就是你的任务。如果 Hibernate 无法标记它以便回滚，那么该事务可能仍旧是 `STATUS_ACTIVE`。在这两种情况下，都要在此工作单元内调用 `UserTransaction#rollback()` 来终止任何已经被发送到数据库的 SQL 语句。

所有的 JPA 操作，包括刷新持久化上下文，都可以抛出 `RuntimeException` 异常。但方法 `UserTransaction#begin()`、`commit()` 甚至 `rollback()` 都会抛出检查过的 `Exception`。该用于回滚的异常需要特殊处理：你希望捕获此异常并且记录它；或者，就是丢失了导致回滚的原始异常。在回滚之后继续抛出原始异常。通常，你的系统中有另一个拦截器层，它最终会

处理该异常，例如通过呈现一个错误界面或联系运营团队来处理。回滚期间的错误更加难以正确处理；我们建议记录日志并且将错误升级，因为失败的回滚表明了一个严重的系统问题。

Hibernate 特性

Hibernate 会抛出类型化异常，帮助你识别错误的 `RuntimeException` 的所有子类型：

- 最常见的 `HibernateException` 是一个一般错误。必须检查异常消息或者通过在异常上调用 `getCause()` 找出更多与起因有关的内容。
- `JDBCException` 是由 Hibernate 的内部 JDBC 层抛出的任意异常。此类异常总是由特定 SQL 语句引起的，并且可以使用 `getSQL()` 得到产生异常的语句。由 JDBC 连接(JDBC 驱动)抛出的内部异常可使用 `getSQLException()` 或 `getCause()` 来得到，并且特定于数据库和供应商的错误代码可以使用 `getErrorCode()` 来得到。
- Hibernate 包括 `JDBCException` 的子类型，以及尝试将由数据库驱动抛出的特定于供应商的错误代码转换成更有意义的内容的内部转换器。该内置转换器可以为 Hibernate 支持的最重要的数据库方言生成 `JDBCConnectionException`、`SQLGrammarException`、`LockAcquisitionException`、`DataException` 和 `ConstraintViolationException`。可以为数据库操作或者强化方言，或者插入一个 `SQLExceptionConverterFactory` 来自定义此转换。

当有些开发人员看到 Hibernate 可以抛出如此多的细粒度异常类型时，他们会很激动。这可能会导致你走错路。例如，你可能会试图捕获 `ConstraintViolationException` 用于验证目的。如果忘记设置 `Item#name` 属性，并且其所映射的列在数据库架构中是 NOT NULL，那么 Hibernate 将在你刷新持久化上下文时抛出这个异常。为什么不捕获它、将(根据错误代码和文本的自定义)失败消息显示给应用程序用户、并且让他们修正该错误呢？这个策略具有两个明显的劣势。

首先，抛出针对数据库的未经检查的值以查看哪个不是可扩展应用程序的合适策略。你至少希望应用层中实现一些数据完整性验证。第二，对于你当前的工作单元来说，异常都是致命的。但这并非应用程序用户解析验证错误的方式：预期它们仍旧位于工作单元内部。围绕这一不匹配的编码是棘手且难以处理的。我们的建议是，使用细粒度异常类型来显示更美观的(致命)错误消息，而非用于验证。例如，可以分别捕获 `ConstraintViolationException` 并且呈现一个显示“应用程序错误：在将数据发送给数据库之前忘记验证数据。请将它报告给开发人员”的界面。对于其他异常，可以呈现一个通用的错误界面。

这样做会在开发期间帮助你，并且也会帮助到任何必须快速确定它是否是应用程序错误(违反约束的、执行的错误 SQL)或数据库系统是否正在加载(无法得到锁)的客户支持工程师。为了进行验证，Bean 验证提供了一个统一的框架。从实体注解中的单个规则集中，Hibernate 可以验证用户接口层的所有域和单行约束，并且可以自动生成 SQL DDL 规则。

你现在知道应该捕获哪些异常以及何时需要它们。你的脑海中可能会出现一个问题：在你已经捕获一个异常并且回滚系统事务之后应该做什么？Hibernate 抛出的异常是致命的。这意味着必须关闭当前的持久化上下文。你无法继续使用抛出异常的 `EntityManager`。

呈现一个错误界面和/或记录该错误，然后让用户使用一个新事务和持久化上下文重启与系统的会话。

像往常一样，这并非是整个全局。有些标准化异常并非致命的：

- `javax.persistence.NoResultException`——在用 `getSingleResult()` 执行 `Query` 或 `TypedQuery` 并且没有从数据库返回任何结果时抛出。可以使用异常处理代码包装该查询调用并且继续使用该持久化上下文。不会标记当前事务用于回滚。
- `javax.persistence.NonUniqueResultException`——在用 `getSingleResult()` 执行 `Query` 或 `TypedQuery` 并且从数据库返回几个结果时抛出。可以使用异常处理代码包装该查询调用并且继续使用该持久化上下文。Hibernate 不会标记当前事务用于回滚。
- `javax.persistence.QueryTimeoutException`——在 `Query` 或 `TypedQuery` 的执行耗时太长时抛出。不标记该事务用于回滚。如果适用的话，你可能希望重复该查询。
- `javax.persistence.LockTimeoutException`——在无法得到悲观锁时抛出。可以在刷新或显式锁定期间发生(本章稍后将更多探讨这个主题)。事务不会被标记用于回滚，并且你可能希望重复该操作。要牢记，已经在努力维持的数据库系统上的不断消耗不会改善该情况。

这个列表上明显缺少的是 `javax.persistence.EntityNotFoundException`。它可以由 `EntityManager#getReference()` 和 `refresh()` 方法以及 `lock()` 抛出，本章稍后将进行介绍。在你尝试访问一个实体实例的引用/代理并且数据库记录不再可用时，Hibernate 就可以抛出它。它是一个致命异常：它会标记当前事务以便回滚，并且必须关闭和丢弃该持久化上下文。

编程式事务分界需要面向像 JTA 的 `UserTransaction` 这样的事务分界接口编写的应用程序代码。另一方面，声明式事务分界不需要额外的编码。

11.1.5 声明式事务分界

在一个 Java EE 应用程序中，可以声明何时希望进入一个事务。然后运行时环境的职责是处理这个问题。通常要在托管组件上(EJB、CDI bean 等)用注解设置事务边界。

可以使用比较老的注解 `@javax.ejb.TransactionAttribute` 在 EJB 组件上声明式地区分事务边界。可以在 18.2.1 节中找到示例。

可以在任何 Java EE 托管的组件上应用较新以及更通用的 `@javax.transaction.Transactional`。可以在 19.3.1 节中找到一个示例。

本章中的其他所有示例都可以在任何 Java SE 环境中运行，无需特殊的运行时容器。因此，从现在开始，在我们专注于特定的 Java EE 应用程序示例之前，你将仅会看到编程式事务分界代码。

接下来，我们专注于 ACID 属性最复杂的方面：如何将并发运行的多个工作单元隔离开来。

11.2 控制并发访问

数据库(及其他事务系统)会尝试确保事务隔离，这意味着，从每个并发事务的角度看，

它表明处理过程中没有其他事务。传统上来说，数据库系统具有带有锁的已实现的隔离。事务可能会在数据库的特定数据项上放置一个锁，暂时避免其他事务对该项的读和/或写访问。有些现代数据库引擎使用多版本并发控制(multiversion concurrency control, MVCC)实现事务隔离，供应商通常会认为这更具可扩展性。我们将通过假设一个锁模型来探讨隔离，但我们大多数的观测也适用于 MVCC。

数据库如何实现并发控制是 Java 持久化应用程序中最重要的方面。应用程序会继承由数据库管理系统提供的隔离保障。例如，Hibernate 不会锁定内存中的一切。如果认为数据库供应商在实现并发控制方面具有多年经验，那么就会看到这种方法的优势。此外，Java 持久化中的一些功能，要么是因为显式使用它们，要么是由于就是如此设计，可以将隔离保障提高到超出数据库所提供的程度。

我们将分几个步骤探讨并发控制。首先，探究最底层：由数据库提供的事务隔离保障。在那以后，你将看到应用程序层用于乐观和悲观并发控制的 Java 持久化功能，以及 Hibernate 可提供何种隔离保障。

11.2.1 理解数据库级别的并发

如果我们正在探讨隔离，那么可以假设两个东西要么是隔离的，要么不是隔离的；现实世界中没有模糊地带。当我们谈论数据库事务时，完整隔离的代价很高。不能在多用户 OLTP 系统中停止一切来独占式访问数据。因此，有几种隔离级别是可用的，它们自然会削弱完整隔离但提高系统性能和可扩展性。

事务隔离问题

首先，查看弱化完全事务隔离时可能出现的几个现象。ANSI SQL 标准会根据允许出现的一些现象来定义标准的事务隔离水平。

如果两个事务都更新一个数据项然后中止第二个事务，就会出现一个丢失的更新，这会造成两个更新都丢失。这会出现于未实现并发控制的系统中，其中没有隔离并发事务。图 11-1 显示了这一情况。

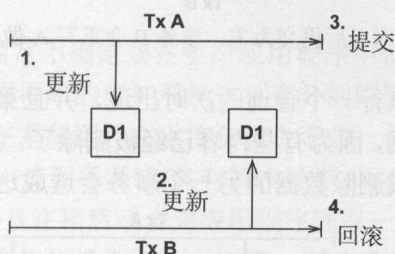


图 11-1 丢失的更新：没有隔离的两个事务更新相同数据

如果一个事务读取由另一个还未提交的事务做出的变更，就会出现脏读取。这是很危险的，因为其他事务做出的变更稍后可能会被回滚，并且无效的数据可能是由第一个事务编写的；参见图 11-2。

如果一个事务读取一个数据项两次并且每次都读取不同的状态，则会出现一个不可重

复的读取。例如，另一个事务可能已经在两次读取之间写入了该数据项并且进行了提交，如图 11-3 所示。

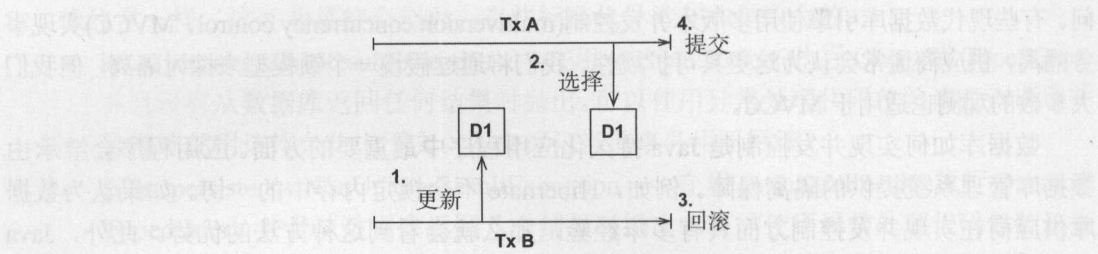


图 11-2 脏读取：事务 A 读取来自事务 B 的未提交数据

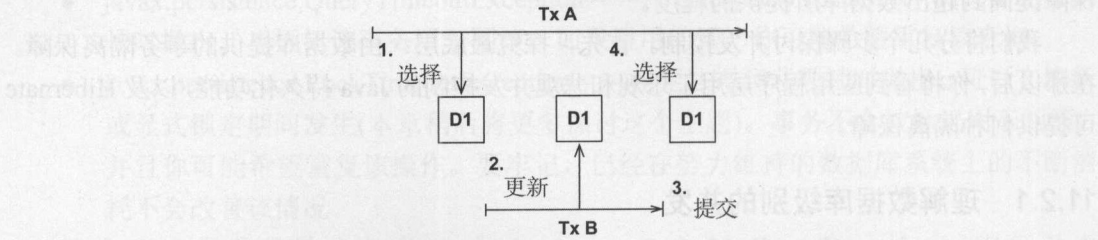


图 11-3 不可重复的读取：事务 A 执行了两次不可重复的读取

不可重复读取的一个特殊用例是后提交为主的问题。想象一下两个并发事务都读取一个数据项的情况，如图 11-4 所示。一个写入它并提交，然后第二个写入它并提交。第一个写入者做出的变更会丢失。这个问题尤其让用户沮丧：用户 A 的变更会被重写而没有任何警告，并且 B 可能已经基于过时的信息做出了决定。

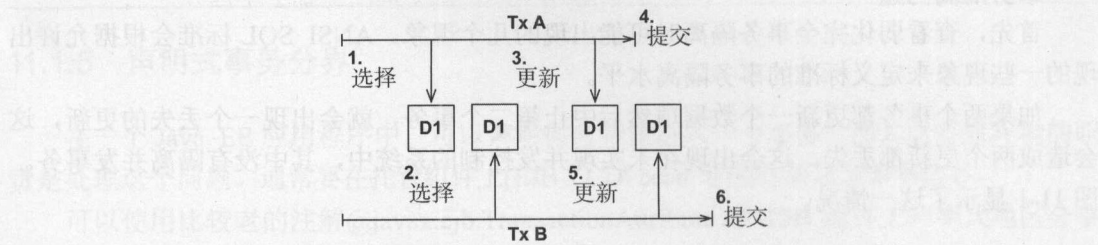


图 11-4 后提交为主：事务 B 会重写 A 做出的变更

据说幻象读会在事务执行一个查询两次时出现，并且第二个结果包含了第一个结果中不可见的数据库或较少的数据，因为有些内容已经被删除了。并不一定需要完全相同的查询。在两个查询执行之间插入或删除数据的另一个事务会造成这种情况，如图 11-5 所示。

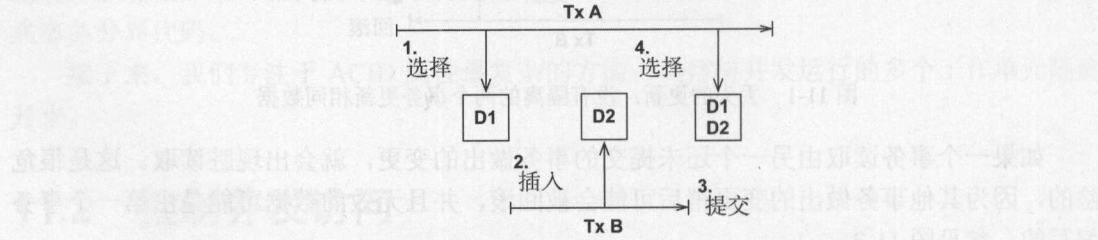


图 11-5 幻象读：事务 A 读取第二个 SELECT 中的新数据

既然你已经理解了所有可能出现的坏情形，那我们就可以定义事务隔离级别并且看看它们能避免哪些问题。

ANSI 隔离级别

标准隔离级别是由 ANSI SQL 标准定义的，但它们并不特定于 SQL 数据库。JTA 正确定义了相同的隔离级别，并且你将使用这些级别来声明你期望的事务隔离。随着提高的隔离级别的成本更高以及性能和可扩展性的严重降低：

- 读未提交的隔离——一个允许脏读取但在读未提交隔离中不丢失更新操作的系统。如果一个未提交事务已经写入了一行，则另一个事务不能写入到该行。不过，任何事务都可以读取任意行。DBMS 可以实现具有独占写入锁的这一隔离级别。
- 读提交的隔离——一个允许不可重复读取而非脏读取的系统会实现读提交的隔离。DBMS 可以通过使用共享读取锁和独占式写入锁来达成这一点。读取事务不会阻止其他事务访问一个行，但未提交的写入事务会阻止所有其他事务访问该行。
- 可重复的读取隔离——一个在可重复读取隔离模式中的系统既不允许不可重复读取，也不允许脏读取。可能会出现幻象读。读取事务会阻止写入事务而不阻止其他读取事务，并且写入事务会阻止所有其他事务。
- 串行化隔离——最严格的隔离会串行化模拟串行执行，就像事务是顺次执行而非并发执行一样。DBMS 不可仅使用行级别的锁来实现串行化。相反，DBMS 必须提供其他一些机制，这些机制要避免新插入行变得对已经执行了会返回该行的查询的事务可见。一个简略的机制是在写入后独占式锁定整个数据库表，因此不会出现任何幻象读。

DBMS 究竟如何实现其锁系统会有显著的不同；每个供应商都有不同的策略。你应该研究 DBMS 的文档以便找出更多与其锁系统、锁如何升级(例如，从行级别到页面级别、到整个表级别)，以及每种隔离级别对系统性能和可扩展性有何影响有关的内容。

知道所有这些技术术语是如何定义的会很好，下面介绍如何帮助你为应用程序选择一种隔离级别？

选择一种隔离级别

开发人员(包括我们自己)通常不确定要在生产应用程序中使用何种事务隔离级别。隔离级别过高会有损高度并发应用程序的可扩展性。不充分的隔离可能会造成应用程序中不易察觉的、难以重现的问题，在系统运行在重度负载之下前，你都不会发现这些问题。

注意，我们会在后续阐释中谈到乐观锁(具有版本控制)，本章稍后会解释这个概念。目前你可能希望跳过这一节，并且在稍后适合为应用程序选取一种隔离级别时再阅读本节。毕竟，选择正确的隔离级别高度依赖于你的特定场景。将以下探讨作为推荐而非亘古不变的真理来阅读。

Hibernate 会努力尝试尽可能对数据库的事务性语意透明。然而，持久化上下文缓存和版本控制会影响这些语意。在 JPA 应用程序中要选择何种合理的数据库隔离级别？

首先，对于几乎所有场景来说，要避免未提交读的隔离级别。在另一个事务中使用一个事务的未提交变更是非常危险的。一个事务的回滚或失败将影响其他并发事务。第一个事务的回滚会让其他事务随之丢失，甚至可能造成它们将数据库变成不正确状态(拍卖商品

的卖家可能被收费两次——与数据库完整性规则一致但并不正确)。可能由一个事务做出的最终被回滚的变更无论如何都会被提交，因为它们可以被读取，然后被另一个成功的事务传递！

其次，大多数应用程序都不需要串行化隔离。幻象读通常没有问题，并且此隔离级别往往难以扩展。很少有已有的应用程序会在生产环境使用串行化隔离，而是依赖选择性应用的在特定情形下有效强制操作串行执行的悲观锁。

接下来，我们思考一下可重复读取。这个级别为数据库事务期间的查询结果集提供了可重复性。这意味着不会从数据库中读提交的更新，如果查询它几次的话。但幻象读仍旧是可行的：可能会出现新的行——如果另一个事务并发提交了这样的变更，则你认为存在的行可能会消失。尽管有时可能希望重复读取，但在每一个事务中通常不需要它们。

JPA 规范假设提交的读是默认的隔离级别。这意味着必须处理不可重复的读取、幻象读以及后提交为主的问题。

我们假设你正在启用域模型实体的版本控制，有些事情 Hibernate 可以为你自动完成。(强制)的持久化上下文缓存和版本控制的组合已经为你提供了可重复性读取隔离的绝佳功能。持久化上下文缓存会确保由一个事务加载的实体实例状态隔离在其他事务所做的变更之外。如果在一个工作单元中检索相同实体实例两次，则第二次查找将在持久化上下文缓存中解决而不会访问数据库。因此，你的读取是可重复的，并且你不会看到冲突的提交数据(不过，你仍将得到幻象读，它们通常非常易于处理)。此外，版本控制会切换到先提交为主。因此，对于几乎所有的多用户 JPA 应用程序来说，启用了实体版本控制，则所有数据库事务的读提交隔离都是可接受的。

Hibernate 会保留数据库连接的隔离级别；它不会变更该级别。大多数产品默认为提交读隔离。可以用几种方式来变更默认的事务隔离级别或者当前事务的设置。

首先，可以检查你的 DBMS 是否在其专属配置中具有一个全局事务隔离级别设置。如果 DBMS 支持标准的 SQL 语句 SET SESSION CHARACTERISTICS，那么就可以执行它以便设置在此特定数据库会话中开启的所有事务的事务设置(这意味着对数据库而非 Hibernate Session 的特定连接)。SQL 还标准化了 SET TRANSACTION 语法，它会设置当前事务的隔离级别。最后，JDBC Connection API 提供了 setTransactionIsolation()方法，它(根据其文档描述)“会试图为此连接变更事务隔离级别。”在一个 Hibernate/JPA 应用程序中，可以从原生的 Session API 中获得一个 JDBC Connection；参阅 17.1 节。

如果正在使用一个 JTA 事务管理器或者一个简单 JDBC 连接池，那么我们推荐另一种方法。JTA 事务管理系统，比如用于本书示例的 Bitronix，允许你为从池中获得的每一个连接设置一个默认事务隔离级别。在 Bitronix 中，可以使用 PoolingDataSource#setIsolationLevel()在启动时设置默认的隔离级别。可以查阅数据源提供程序、应用程序服务器或者 JDBC 连接池的文档以获得更多信息。

我们假设从现在开始数据库连接默认为提交读隔离级别。应用程序中的特定工作单元可能经常需要一个不同的、通常更为严格的隔离级别。相较于变更整个事务的隔离级别，应该使用 Java 持久化 API 获得相关数据上的附加锁。此细粒度锁在高并发的应用程序中更加可扩展。JPA 提供了乐观版本检查和数据库级别的悲观锁。

11.2.2 乐观并发控制

当并发修改很少并且稍后在工作单元中检测冲突是可行的时候，以乐观的方式处理并发就是合适的。JPA 提供了作为乐观冲突检测过程的自动版本检查。

首先要启用版本控制，因为它默认是关闭的——这就是你什么都不做时会得到后提交为主的原因。大多数多用户应用程序，尤其是 Web 应用程序，都应该借助用于所有并发修改的 `@Entity` 实例的版本控制，以启用对用户更友好的先提交为主。

前面几节已经有些枯燥乏味；是时候开始编码了。在启用自动版本检查之后，你将看到手动版本检查如何生效以及何时必须使用它。

启用版本控制

在实体类的特殊附加属性上使用 `@Version` 注解启用版本控制，如代码清单 11.2 所示。

代码清单 11.2 在映射的实体上启用版本控制

路径：/model/src/main/java/org/jpwh/model/concurrency/version/Item.java

```
@Entity
public class Item implements Serializable {

    @Version
    protected long version;

    // ...
}
```

在这个示例中，每个实体实例都带有一个数值的版本。它被映射到 ITEM 数据库表的一个额外的列；像往常一样，该列名称默认为属性名称，这里是 VERSION。该属性和列的实际名称无关紧要——如果在 DBMS 中 VERSION 是一个保留关键字，那么可以重命名它。

可以将一个 `getVersion()` 方法添加到这个类，但不应该使用设置方法，并且应用程序不应该修改这个值。Hibernate 会自动修改该版本值：在持久化上下文刷新期间发现一个 Item 实例变脏时，它会递增该版本号。该版本是一个简单的计数器，没有任何有用的超出并发控制之外的语义值。可以使用 `int`、`Integer`、`short`、`Short` 或 `Long` 来代替 `long`；如果该版本号达到了数据类型的限制，则 Hibernate 会进行包装并且再次从零开始。

在刷新期间递增了检测到的脏 Item 的版本号之后，Hibernate 会在执行 UPDATE 和 DELETE SQL 语句时进行版本比较。例如，假设在一个工作单元中，你加载了一个 Item 并且修改了其名称，如代码清单 11.3 所示。

代码清单 11.3 Hibernate 自动递增和检查版本

路径：/examples/src/test/java/org/jpwh/test/concurrency/Versioning.java

```
tx.begin();
em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID); ← ①通过标识符检索
// select * from ITEM where ID = ?
```

```

assertEquals(item.getVersion(), 0);
item.setName("New Name");
em.flush();
// update ITEM set NAME = ?, VERSION = 1 where ID = ? and VERSION = 0

```

← ②实例版本: 0

← ③刷新持久化上下文

- ① 通过标识符检索一个实体实例会用一个 SELECT 从数据库中加载当前版本。
- ② 该 Item 实例的当前版本是 0。
- ③ 在刷新持久化上下文时, Hibernate 会检测脏 Item 实例并且将其版本递增到 1。SQL UPDATE 现在会执行版本检查, 将新的版本保存到数据库, 但仅在数据库版本仍然是 0 的时候才这样做。

关注该 SQL 语句, 特别是 UPDATE 及其 WHERE 子句。此更新仅在数据库中有一个 VERSION = 0 的行时才会成功。JDBC 会将所更新行的编号返回给 Hibernate; 如果结果是零, 就意味着该 ITEM 行要么不存在了, 要么不再具有版本 0 了。Hibernate 会在刷新期间检测此冲突, 并且抛出 `javax.persistence.OptimisticLockException` 异常。

现在思考两个用户在相同时间执行此工作单元的情况, 如之前图 11-4 中所示。第一个执行提交的用户会更新 Item 的名称并且将递增的版本 1 刷新到数据库。第二个用户的刷新(和提交)会失败, 因为他们的 UPDATE 语句无法在数据库中找到版本为 0 的行。该数据库版本是 1。因此, 先提交为主, 并且可以捕获 `OptimisticLockException` 并且对其进行特别处理。例如, 可以将以下消息显示给第二个用户: “你使用的数据已经被其他人修改过了。请用新数据重新开始你的工作单元。单击 Restart 按钮开始处理。”

哪些修改会触发实体版本的递增呢? 无论何时实体实例变脏, Hibernate 都会递增该版本。这包括实体的所有脏值类型属性, 无论它们是否是单值的(比如 String 或 int 属性)、嵌入的(比如 Address)或者集合。已经用 `mappedBy` 变成只读的 `@OneToMany` 和 `@ManyToMany` 关联集合是例外。对这些集合添加或移除元素不会递增其所属实体实例的版本号。你应该知道, 这些在 JPA 中并非是标准化的——在访问一个共享数据库时不要依赖两个实现相同规则的 JPA 提供程序。

用共享数据库进行版本控制

如果有几个应用程序访问你的数据库, 并且它们并非都使用 Hibernate 的版本控制算法, 那么就会遇到并发性问题。一个简单的解决方案是使用数据库级别的触发器和存储过程: INSTEAD OF 触发器可以在进行任何 UPDATE 时执行存储过程; 它会替代该更新来运行。在该存储过程中, 可以检查应用程序是否递增的行的版本; 如果版本没有被更新或者更新中没有包含版本列, 那么就会知道该语句不是由 Hibernate 应用程序发出的。然后可以在应用 UPDATE 之前在存储过程中递增该版本。

如果不希望在特定属性值被修改时递增实体实例的版本, 则要用 `@org.hibernate.annotations.OptimisticLock(excluded = true)` 注解该属性。你可能不希望数据库架构中存在额外的 VERSION 列。或者, 你的实体类上可能已经有了一个“最后一次更新”的时间戳属性以及一个匹配的数据库列了。Hibernate 可以使用时间戳代替额外的计数器字段来检查版本。

使用时间戳进行版本控制

如果数据库架构已经包含了一个时间戳列，比如 LASTUPDATED 或 MODIFIED_ON，那么可以映射它以便于自动版本检查，而不是使用一个数值计数器。见代码清单 11.4。

代码清单 11.4 启用时间戳版本控制

路径: /model/src/main/java/org/jpwh/model/concurrency/versiontimestamp/Item.
java

@Entity

```
public class Item {
```

@Version

```
// Optional: @org.hibernate.annotations.Type(type = "dbtimestamp")  
protected Date lastUpdated;
```

```
// ...  
}
```

这个示例将列 LASTUPDATED 映射到一个 java.util.Date 属性；Calendar 类型也适用于 Hibernate。JPA 标准没有定义版本属性的这些类型；JPA 仅认为 java.sql.Timestamp 是可移植的。这样做没什么吸引力，因为必须将 JDBC 类导入到域模型中。应该尝试将像 JDBC 这样的实现细节保持在域模型类之外，这样就能在尽可能多的环境中对其进行测试、实例化、跨编译(例如用 GWT 跨到 JavaScript)、序列化和反序列化。

从理论上说，使用时间戳的版本控制稍微有些不安全，因为在相同的毫秒内两个并发事务都可能加载和更新相同 Item；JVM 通常不具有毫秒级精度，这一实际情况会加剧这个问题(为了得到确保的精确度，应该查阅 JVM 和操作系统文档)。此外，从 JVM 检索当前时间在一个群集环境中也不一定安全，其中节点的系统时间可能不是同步的，或者时间同步并非像你的事务负荷所需的那样精确。

Hibernate 特性

可以通过在版本属性上放置一个 @org.hibernate.annotations.Type(type="dbtimestamp") 注解来切换为从数据库服务器检索当前时间。Hibernate 现在会在更新之前询问数据库当前时间是什么，比如在 H2 上使用 call current_timestamp()。这就为你提供了用于同步的单一时间源。并非所有的 Hibernate SQL 方言都支持这一点，所以要检查你配置的方言的源以及它是否重写了 getCurrentTimestampSelectString() 方法。此外，每次递增总是会有访问数据库的开销。

我们建议，新的项目借助数值计数器而非时间戳进行版本控制。如果正在处理一个遗留数据库架构或已有的 Java 类，那么可能就无法引入一个版本或者时间戳属性和列。如果是这样的话，Hibernate 为你提供了一个可选策略。

Hibernate 特性

不使用版本号或时间戳进行版本控制

如果没有版本或时间戳列，Hibernate 也还是可以执行自动的版本控制。版本控制的这

一替代实现会在 Hibernate 检索实体实例时(或者最后一次刷新持久化上下文时)针对持久化属性的未修改值检查当前的数据库状态。

你要使用 Hibernate 专有的注解 `@org.hibernate.annotations.OptimisticLocking` 来启用这一功能:

路径: `\model\src\main\java\org\jpwh\model\concurrency\versionall\Item.java`

```
@Entity
@org.hibernate.annotations.OptimisticLocking(
    type = org.hibernate.annotations.OptimisticLockType.ALL)
@org.hibernate.annotations.DynamicUpdate
public class Item {

    // ...

}
```

对于此策略,也必须使用 `@org.hibernate.annotations.DynamicUpdate` 来启用 UPDATE 语句的动态 SQL 生成,就像 4.3.2 节中所阐释的一样。

Hibernate 现在会执行以下 SQL 来刷新一个 Item 实例的修改:

```
update ITEM set NAME = 'New Name'
where ID = 123
    and NAME = 'Old Name'
    and PRICE = '9.99'
    and DESCRIPTION = 'Some item for auction'
    and ...
    and SELLER_ID = 45
```

Hibernate 在 WHERE 子句中列出了所有的列及其最后的已知值。如果当前事务已经修改了这些值中的任何一个或是删除了该行,那么这个语句就会返回零个已更新行。然后 Hibernate 会在刷新时抛出一个异常。

另外,如果切换到 `OptimisticLockType.DIRTY`,那么 Hibernate 仅会在约束中包含修改过的属性(本示例中仅有 NAME)。这意味着两个工作单元可以并发修改相同的 Item,而 Hibernate 仅在它们都修改相同的值类型属性(或外键值)时才检测冲突。上一个 SQL 片段的 WHERE 子句会缩减为 `where ID = 123 and NAME = 'Old Name'`。其他人可以并发修改价格,而 Hibernate 不会检测任何冲突。只有在应用程序并发修改名称时,你才会得到一个 `javax.persistence.OptimisticLockException` 异常。

在大多数情况下,仅检查脏属性并不是用于业务实体的一个好策略。如果修改了描述,那么变更一个单品的价格可能就不合适了!

这个策略也不适用于分离实体和合并:如果将一个分离实体合并到一个新的持久化上下文中,那就无法获知“旧”值。该分离的实体实例将必须携带一个版本号或者时间戳以便用于乐观并发控制。

在两个并发事务试图提交相同数据块上的修改时,Java 持久化中的自动版本控制会避免更新丢失。版本控制还可以帮助你在需要时手动获得额外的隔离保障。

手动版本检查

这里是一个需要可重复性数据库读取的场景：我们假设你的拍卖系统中有一个类别并且每个 Item 都属于一个 Category。这是一个 Item#category 实体关联的常规@ManyToOne 映射。

我们假设你希望合计几个类别中所有商品的价格。这就需要有一个对每个类别的所有商品的查询，以便累加价格。问题在于，如果当仍旧在查询并且遍历所有的类别和商品时有人将一个 Item 从一个 Category 移动到另一个 Category，会发生什么呢？使用提交读隔离，当程序运行时，相同的 Item 可能会出现两次！

为了让“得到每个类别中的商品”读取可重复性，JPA 的 Query 接口提供了一个 setLockMode() 方法。看看代码清单 11.5 中的程序。

代码清单 11.5 在刷新时请求一个版本检查以确保可重复读取

路径：/examples/src/test/java/org/jpwh/test/concurrency/Versioning.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();
BigDecimal totalPrice = new BigDecimal(0);
for (Long categoryId : CATEGORIES) {
    List<Item> items =
        em.createQuery("select i from Item i where i.category.id = :catId")
            .setLockMode(LockModeType.OPTIMISTIC)
            .setParameter("catId", categoryId)
            .getResultList();

    for (Item item : items)
        totalPrice = totalPrice.add(item.getBuyNowPrice());
}

tx.commit();
em.close();

assertEquals(totalPrice.toString(), "108.00");
```

- ① 对于每一个 Category，都使用一个 OPTIMISTIC 锁模式查询所有的 Item 实例。Hibernate 现在知道它必须在刷新时检查每个 Item。
- ② 对于每个之前使用锁查询加载的 Item，Hibernate 都会在刷新时执行一个 SELECT。它会检查每个 ITEM 行的数据库版本是否仍然与加载时一样。如果任何 ITEM 行具有不同的版本或者该行不再存在，则会抛出一个 OptimisticLockException 异常。

不要被锁定算法混淆了：JPA 规范没有规定每个 LockModeType 究竟如何实现；对于 OPTIMISTIC，Hibernate 会执行版本检查。没有实际的锁会参与进来。必须像之前说明的那样在 Item 实体类上启用版本控制；否则，就无法在 Hibernate 中使用乐观 LockModeTypes。

Hibernate 不会为手动版本检查批量执行或者优化 SELECT 语句：如果合计 100 个商品，则会在刷新时得到 100 个额外的查询。就像我们将在本章稍后介绍的那样，悲观方法对于这个特定用例来说可能是一个更好的解决方案。

常见问题解答：为何持久化上下文缓存不能避免这个问题？

“得到特定类别的所有商品”查询会在一个 `ResultSet` 中返回商品数据。然后 Hibernate 会查看此数据中的主键值并且首先尝试在持久化上下文缓存中解析每个 `Item` 的剩余详细信息——它会检查是否已经用该标识符加载了一个 `Item` 实例。不过，这个缓存在示例程序中无法使用：如果并发事务将一个商品移动到另一个类别，则可能会在不同的 `ResultSet` 中多次返回该商品。Hibernate 将执行其持久化上下文查找并且提示，“我已经加载了该 `Item` 实例；我们使用内存中已经存在的东西吧。” Hibernate 甚至不会意识到为商品指定的类别已经被修改或者该商品再次出现在一个不同的结果中了。因此这就是持久化上下文的可重复性读取功能隐藏并发提交数据的情况。需要手动检查版本以便在预期数据没有变更的情况下弄明白数据是否发生了变更。

正如之前示例所示的那样，`Query` 接口接受 `LockModeType`。使用相同的 `setLockMode()` 方法，`TypedQuery` 和 `NamedQuery` 接口也支持显式锁模式。

JPA 中提供了一个额外的乐观锁模式，强制递增一个实体的版本。

强制版本递增

如果两个用户同时对相同拍卖出价会发生什么？当一个用户做出新的出价时，应用程序必须做几件事情：

(1) 从数据库中检索 `Item` 的当前最高 `Bid`。

(2) 用最高 `Bid` 对比新的 `Bid`；如果新的 `Bid` 更高，则必须将其存储在数据库中。

这两个步骤之间可能存在竞争状态。如果在读取最高 `Bid` 和放置新 `Bid` 之间，另一个 `Bid` 出现，那么你将不会看到它。这一冲突不可见；甚至启用 `Item` 的版本控制也于事无补。该 `Item` 不会在程序运行期间被修改。强制 `Item` 的版本递增会检测到该冲突。见代码清单 11.6。

代码清单 11.6 强制一个实体实例的版本递增

路径：/examples/src/test/java/org/jpwh/test/concurrency/Versioning.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();
Item item = em.find(
    Item.class,
    ITEM_ID,
    LockModeType.OPTIMISTIC_FORCE_INCREMENT
);
Bid highestBid = queryHighestBid(em, item);
try {
    Bid newBid = new Bid(
        new BigDecimal("44.44"),
        item,
        highestBid
    );
    em.persist(newBid);
} catch (InvalidBidException ex) { }
```

①告知 Hibernate 递增 Item 版本

②持久化 Bid 实例

③检查出价


```
tx.commit();
em.close();
```

← ④插入出价

- ❶ find()接受 LockModeType。OPTIMISTIC_FORCE_INCREMENT 模式告知 Hibernate 被检索 Item 的版本应该在加载后递增，即便在工作单元中它永远不会被修改也应如此。
- ❷ 该代码持久化了一个新的 Bid 实例；这不会影响 Item 实例的任何值。一个新行会被插入到 BID 表中。Hibernate 不会在没有 Item 的强制版本递增的情况下并发检测做出的出价。
- ❸ 要使用一个经检查的异常来验证新的出价金额。它必须比当前最高出价大。
- ❹ 在刷新该持久化上下文时，Hibernate 会为新的 Bid 执行一个 INSERT 并且用版本检查强制 Item 的 UPDATE。如果有人并发修改了该 Item 或者在这个过程中并发放置了一个 Bid，则 Hibernate 会抛出一个异常。

对于该拍卖系统来说，并发放置出价肯定是一个频繁操作。在许多情况下，当插入或修改数据并且希望递增聚合的一些根实例的版本时，手动递增一个版本是很有用的。

注意，如果使用 Item#bids 的 @ElementCollection 而不是使用具有 @ManyToOne 的 Bid#item 实体关联，那么将 Bid 添加到集合将递增该 Item 版本。强制的递增就没有必要了。你可能希望回顾 7.3 节中父/子歧义的探讨以及聚合和组合如何在 ORM 中运行。

到目前为止，我们已经重点介绍了乐观并发控制：我们预期并发修改较少发生，因此不会阻止并发访问并且延迟检测冲突。有时你知道冲突将频繁发生，并且希望在某些数据上放置一个独占锁。这就需要悲观方法。

11.2.3 显式悲观锁

我们用一个悲观锁而非乐观版本检查来重复“手动版本检查”一节中所示的过程。你要再次汇总几个类别中所有商品的总价。这之前代码清单 11.5 中所示的代码相同，具有一个不同的 LockModeType。见代码清单 11.7。

代码清单11.7 悲观锁定数据

路径：/examples/src/test/java/org/jpwh/test/concurrency/Locking.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

BigDecimal totalPrice = new BigDecimal(0);
for (Long categoryId : CATEGORIES) {
    List<Item> items =
        em.createQuery("select i from Item i where i.category.id = :catId")
            .setLockMode(LockModeType.PESSIMISTIC_READ)
            .setHint("javax.persistence.lock.timeout", 5000)
            .setParameter("catId", categoryId)
            .getResultList();
    for (Item item : items)
```

①查询所有的 Item 实例

②查询成功意味着独占锁

```

        totalPrice = totalPrice.add(item.getBuyNowPrice());
    }
    tx.commit();
    em.close();

```

← ③ 释放锁

```

assertEquals(totalPrice.compareTo(new BigDecimal("108")), 0);

```

- ❶ 对于每一个 Category，都在 PESSIMISTIC_READ 锁模式中查询所有 Item 实例。Hibernate 会使用该 SQL 查询锁定数据库中的行。如果另一个事务持有一个冲突锁，则等待 5 秒。如果无法获得该锁，则查询会抛出一个异常。
- ❷ 如果查询成功返回，那么知道数据上保持有一个独占锁并且没有其他事务可以用独占锁访问它或者在这个事务提交之前修改它。
- ❸ 在事务完成时，你的锁会在提交之后被释放。

JPA 规范定义，锁模式 PESSIMISTIC_READ 会确保可重复读取。JPA 还标准化了 PESSIMISTIC_WRITE 模式，它具有额外的保障：除了可重复读取外，JPA 提供程序必须串行化数据访问，并且不能出现幻象读。

是否实现这些要求取决于 JPA 提供程序。对于这两个模式来说，Hibernate 会在加载数据时将一个“for update”的子句附加到 SQL 查询。这会在数据库级别将锁放置在行上。Hibernate 使用何种锁取决于 LockModeType 以及你的 Hibernate 数据库方言。

例如，在 H2 上，该查询是 SELECT * FROM ITEM ... FOR UPDATE。因为 H2 仅支持一种类型的独占锁，所以 Hibernate 会为所有的悲观模式生成相同的 SQL。

另一方面，PostgreSQL 支持共享读取锁：PESSIMISTIC_READ 模式会将 FOR SHARE 附加到 SQL 查询。PESSIMISTIC_WRITE 会使用一个具有 FOR UPDATE 的独占写入锁。

在 MySQL 上，PESSIMISTIC_READ 转译为 LOCK IN SHARE MODE，而 PESSIMISTIC_WRITE 转译为 FOR UPDATE。检查数据库方言。这是使用 getReadLockString() 和 getWriteLockString() 方法配置的。

JPA 中悲观锁的持续期间是单个数据库事务。这意味着无法使用独占锁来阻止比单个数据库事务更长的并发访问。当无法获得数据库锁时，会抛出一个异常。将其与乐观方法对比，其中 Hibernate 会在提交时而不是查询时抛出一个异常。使用悲观策略，就会知道当锁定查询成功时可以安全地读取和写入数据。使用乐观方法，当提交时，你做最好的期望，然后可能会出乎你意料。

离线锁

悲观数据库锁仅保留用于单个事务。其他的锁实现也是可行的：例如，保持在内存中的锁，或者数据库中所谓的锁定表。这些种类锁的通用名称是离线锁。

比单个数据库事务更长的悲观锁通常是一个性能瓶颈；每个数据访问都涉及额外的对全局同步的锁管理器的锁检查。不过，乐观锁是用于长期运行会话的完美并发控制策略（下一章将会介绍）并且能很好地执行。根据冲突-解决策略——检测到冲突后做什么——应用程序用户可能会像被阻止的并发访问一样感到满意。他们也可能赞赏应用程序在其他人查看相同数据时没有将他们锁定在特定界面之外。

可以配置数据库等待获得锁的时长并且使用 javax.persistence.lock.timeout 提示阻止毫

秒级查询。就像以往的提示一样，Hibernate 可能会忽略它，这取决于数据库产品。例如，H2 不支持每个查询的锁超时，仅支持每个连接的全局锁超时(默认为 1 秒)。使用一些方言，比如 PostgreSQL 和 Oracle，0 这个锁超时会将 NOWAIT 子句附加到 SQL 字符串。

我们已经显示了应用到一个 Query 的锁超时提示。还可以为 find()操作设置超时提示：

路径：/examples/src/test/java/org/jpwh/test/concurrency/Locking.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Map<String, Object> hints = new HashMap<String, Object>();
hints.put("javax.persistence.lock.timeout", 5000);

Category category =
    em.find(
        Category.class,
        CATEGORY_ID,
        LockModeType.PESSIMISTIC_WRITE,
        hints
    );

category.setName("New Name");

tx.commit();
em.close();
```

← 如果方言支持，则执行 SELECT ..
FOR UPDATE WAIT 5000

当无法获得一个锁时，Hibernate 会抛出 `javax.persistence.LockTimeoutException` 或者 `javax.persistence.PessimisticLockException` 异常。如果 Hibernate 抛出一个 `PessimisticLockException` 异常，则事务必须回滚，并且该工作单元结束。另一方面，超时异常对于事务并非致命，就像 11.1.4 节中所阐释的一样。Hibernate 会抛出哪个异常还是取决于 SQL 方言。例如，由于 H2 不支持每个语句的锁超时，因此总是会得到一个 `PessimisticLockException` 异常。

可以同时使用 `PESSIMISTIC_READ` 和 `PESSIMISTIC_WRITE` 锁模式，即便没有启用实体版本控制也行。它们会转译成具有数据库级别锁的 SQL 语句。

不过，`PESSIMISTIC_FORCE_INCREMENT` 这个特殊模式需要版本化的实体。在 Hibernate 中，这个模式会执行一个 `FOR UPDATE NOWAIT` 锁(或者任何你的方言支持的东西；检查它的 `getForUpdateNowaitString()` 实现)。然后，在查询返回之后，Hibernate 会立即递增版本并且 `UPDATE (!)` 每一个返回的实体实例。这就像所有并发事务表明你已经更新了这些行，即便你目前还没有修改任何数据。这个模式很少用到，主要是用于“强制版本递增”一节中阐释的聚合锁定。

READ 和 WRITE 锁模式

有一些来自 Java 持久化 1.0 的比较旧的锁模式，你不应再使用它们。`LockModeType.READ` 等效于 `OPTIMISTIC`，而 `LockModeType.WRITE` 等效于 `OPTIMISTIC_FORCE_INCREMENT`。

如果启用悲观锁,则 Hibernate 仅会锁定对应于实体实例状态的行。换句话说,如果锁定一个 Item 实例,则 Hibernate 会在 ITEM 表中锁定它的行。如果使用联合的继承映射策略,则 Hibernate 将识别它并且锁定超表和子表中的适当行。这同样适用于实体的任何辅助表映射。因为 Hibernate 会锁定所有行,任何外键位于该行中的关系也都会被实际锁定:如果 SELLER_ID 外键列位于 ITEM 表中,则会锁定 Item#seller 关联。不会锁定实际的 Seller 实例!集合或 Item 的其他关联中的外键都不会位于其他表中。

在 DBMS 中使用独占锁,就可以看到事务失败,因为你碰到了死锁的情况。

扩展锁的范围

JPA 2.0 定义了 PessimisticLockScope.EXTENDED 选项。它可以被设置为具有 javax.persistence.lock.scope 的查询提示。如果启用了的话,持久化引擎就会将锁定数据的范围扩展为包含集合和锁定实体的关联联结表中的所有数据。在编写本书时, Hibernate 还没有实现此功能。

11.2.4 避免死锁

如果 DBMS 依赖独占锁来实现事务隔离,就会出现死锁。思考以下工作单元,按特定顺序更新两个 Item 实体实例:

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item itemOne = em.find(Item.class, ITEM_ONE_ID);
itemOne.setName("First new name");

Item itemTwo = em.find(Item.class, ITEM_TWO_ID);
itemTwo.setName("Second new name");

tx.commit();
em.close();
```

Hibernate 会在刷新持久化上下文时执行两个 SQL UPDATE 语句。第一个 UPDATE 会锁定表示 Item 1 的行,而第二个 UPDATE 会锁定 Item 2:

```
update ITEM set ... where ID = 1;    ← 锁定行 1
update ITEM set ... where ID = 2;    ← 尝试锁定行 2
```

如果一个具有相反 Item 更新顺序的类似程序在并发事务中执行,就可能会(也可能不会!)出现死锁:

```
update ITEM set ... where ID = 2;    ← 锁定行 2
update ITEM set ... where ID = 1;    ← 尝试锁定行 1
```

出现一个死锁,那么这两个事务都会被阻止并且无法继续执行,每个事务都要等待锁被释放。死锁的几率通常很小,但在高并发的应用程序中,两个 Hibernate 应用程序都可能执行此种交替更新。注意可能不会在测试期间看到死锁(除非你编写正确类型的测试)。当生产环境中的应用程序必须处理高事务负荷时,死锁就可能突然出现。通常 DBMS 会在超

时期限和失败之后终止其中一个死锁的事务；然后其他事务可以继续执行。或者，视 DBMS 而定，DBMS 可以自动检测死锁的情况并且立即终止其中一个事务。

你应该尝试避免事务失败，因为它们难以从应用程序代码中恢复。一个解决方案是在更新单行锁定整个表时以串行化模式运行数据库连接。并发事务必须等待首个事务完成其工作。另外，当对数据执行 SELECT 操作时，第一个事务可以在所有数据上获得一个独占锁，如上一节中所示。然后任何并发事务也必须等待这些锁的释放。

一个可选的显著降低死锁概率的实用优化是通过主键值排序 UPDATE 语句：Hibernate 应该总是在更新行 2 之前更新具有主键 1 的行，无论应用程序加载和修改数据的顺序如何。可以用配置属性 `hibernate.order_updates` 为整个持久化单元启用这一优化。然后 Hibernate 会根据刷新期间检测到的所修改的实体实例和集合元素的主键值按照升序对其执行的所有 UPDATE 语句排序(正如之前提到过的，要确保你完全理解 DBMS 产品的事务和锁定行为。Hibernate 从 DBMS 中继承了其大多数事务保障；例如，你的 MVCC 数据库产品可能避免了读取锁但很可能要依赖独占锁用于写隔离，并且你可能会看到死锁)。

我们还没有机会介绍 `EntityManager#lock()` 方法。它接受一个已经加载的持久化实体实例和一个锁模式。它会执行你在使用 `find()` 和一个 Query 时看到过的相同锁定，只不过它不会加载该实例。此外，如果一个版本化的实体被悲观锁定，那么 `lock()` 方法就会在数据库上执行一个即时版本检查并且可能会抛出一个 `OptimisticLockException` 异常。如果该数据库表示形式不再存在，则 Hibernate 会抛出一个 `EntityNotFoundException` 异常。最后，`EntityManager#refresh()` 方法也接受一个锁模式，它具有相同的语义。

我们现在已经介绍了最低级别的数据库的并发控制以及 JPA 的乐观锁和悲观锁功能。我们仍旧有并发的另一个方面需要探讨：在事务之外访问数据。

11.3 非事务性数据访问

JDBC Connection 默认处于自动提交模式。此模式对于执行专门的 SQL 很有用。

想象一下，你要使用一个 SQL 控制台连接你的数据库，并且运行一些查询，甚至可能要更新和删除一些行。这一交互式数据访问是专用的；大多数考虑使用工作单元的时候你都没有一个计划或一系列语句。数据库连接上默认的自动提交模式对于此类数据访问来说是完美的——毕竟，你不希望为每个编写和执行的 SQL 语句输入 `begin transaction` 和 `end transaction`。在自动提交模式中，一个(短)数据库事务会为每个发送到数据库的 SQL 语句开始和结束。你在非事务性模式中的工作是高效的，因为对于与 SQL 控制台的会话来说没有原子性或隔离性保障(唯一的保障是单个 SQL 语句是原子的)。

根据定义，一个应用程序总是会执行计划好的一系列语句。因此总是创建事务边界来将你的语句分组组成原子且相互隔离的单元，这看起来是合理的。不过，在 JPA 中，特殊的行为是与自动提交模式相关联的，并且可能需要它来实现长期运行的会话。可以在自动提交模式中访问数据库和读取数据。

11.3.1 在自动提交模式中读取数据

思考一下代码清单 11.8 所示的示例，它会加载一个 `Item` 实例，修改其名称，然后通过刷新回滚该变更。

代码清单 11.8 在自动提交模式中读取数据

路径: `/examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java`

```
EntityManager em = JPA.createEntityManager(); ← ❶ 非同步的模式

Item item = em.find(Item.class, ITEM_ID); ← ❷ 访问数据库
item.setName("New Name");

assertEquals( ← ❸ 返回原始值
    em.createQuery("select i.name from Item i where i.id = :id")
      .setParameter("id", ITEM_ID).getSingleResult(),
    "Original Name"
);

assertEquals( ← ❹ 返回已经加载的实例
    ((Item) em.createQuery("select i from Item i where i.id = :id")
      .setParameter("id", ITEM_ID).getSingleResult()).getName(),
    "New Name"
);

// em.flush(); ← ❺ 抛出异常

em.refresh(item); ← ❻ 回滚变更
assertEquals(item.getName(), "Original Name");

em.close();
```

- ❶ 当创建 `EntityManager` 时，没有活动的事务。现在持久化上下文就处于特殊的非同步模式；Hibernate 不会自动刷新。
- ❷ 可以访问数据库来读取数据；这个操作会执行一个 `SELECT`，在自动提交模式中发送到数据库。
- ❸ 通常 Hibernate 会在执行一个 `Query` 时刷新持久化上下文。但由于该上下文是非同步的，因此刷新并不会发生，并且该查询会返回旧的、原始的数据库值。具有标量结果的查询不是可重复的：你会看到数据库中以及在 `ResultSet` 中提供给 Hibernate 的任何值。如果处于同步模式，那么这也不是可重复的读取。
- ❹ 检索一个托管的实体实例涉及在当前持久化上下文中 JDBC 结果集排列期间的一个查找。该具有已修改名称的已经加载的 `Item` 实例是从持久化上下文中返回的；来自数据库的值会被忽略。这是一个实体实例的可重复读取，甚至无需系统事务。
- ❺ 如果试图手动刷新该持久化上下文，以便存储新的 `Item#name`，那么 Hiberante 就会抛出一个 `javax.persistence.TransactionRequiredException` 异常。无法在非同步模式中执行一个 `UPDATE`，因为不能回滚该变更。

- ⑥ 可以使用 `refresh()` 方法回滚你做出的变更。它会从数据库中加载当前的 `Item` 状态并且重写内存中做出的变更。

有了非同步的持久化上下文，就可以在自动提交模式中使用 `find()`、`getReference()`、`refresh()` 或查询读取数据。也可以按需加载数据：如果访问代理，它们就会被初始化，并且如果开始遍历其元素，就会加载集合。但如果尝试刷新该持久化上下文或者使用除 `LockModeType.NONE` 之外的其他模式锁定数据，则会出现 `TransactionRequiredException` 异常。

到目前为止，自动提交模式看起来还没太多用处。实际上，许多开发人员通常都是出于错误的原因而借助自动提交：

- 许多小的每语句的数据库事务(这就是自动提交的含义)都不会提升应用程序的性能。
- 你不会提高应用程序的可扩展性：相较于用于每个 SQL 语句的许多小事务，一个较长期运行的数据库事务可以长期地持有数据库锁。这是一个小问题，因为 `Hibernate` 会在事务中尽可能迟地写入数据库(在提交时刷新)，所以数据库已经短时间地持有了写入锁。
- 如果应用程序并发修改了数据，那么就会得到较弱的隔离保障。基于读取锁的可重复读取在自动提交模式下是不可行的(自然，持久化上下文缓存在这里是有用的)。
- 如果 DBMS 具有 MVCC(比如 `Oracle`、`PostgreSQL`、`Informix` 和 `Firebird`)，那么你可能就希望将其功能用于快照隔离，以避免不可重复的读取和幻象读。每个事务都会得到其自己的数据私有快照；你只会看到数据的一个(数据库内部)版本，与它在事务开始之前的版本一样。使用自动提交模式，快照隔离就毫无意义了，因为没有事务的作用域。
- 你的代码将更加难以理解。现在代码的任何阅读者都必须特别关注持久化上下文是否联结了一个事务，或者它是否处于非同步模式。如果总是在系统事务中分组操作，那么即便只是读取数据，每个人都可以遵循此简单规则，并且难以找到并发性问题的可能性将会降低。

因此，非同步持久化上下文的好处是什么呢？如果刷新没有自动发生，那么可以准备好对事务之外的修改进行排队。

11.3.2 对修改进行排队

以下示例会用非同步的 `EntityManager` 存储一个新的 `Item` 实例：

路径：/examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java

```
EntityManager em = JPA.createEntityManager();
```

```
Item newItem = new Item("New Item");
```

```
em.persist(newItem)
```

← 保存瞬时实例

```
assertNotNull(newItem.getId());
```

```
tx.begin();
```

← 存储变更

```
if (!em.isJoinedToTransaction())
```

```
em.joinTransaction();
```

```
tx.commit();
```

```
em.close();
```

 刷新

- ① 可以调用 `persist()` 保存具有非同步持久化上下文的瞬时实体实例。Hibernate 只会提取一个新的标识符值，通常是通过调用一个数据库序列来实现，并且将它分配给实例。上下文中该实例现在处于持久化状态，但 SQL INSERT 还没有发生。注意，这仅在使用预先插入的标识符生成器时才可行；参阅 4.2.5 节。

- ② 当准备好存储变更时，要将持久化上下文联结到一个事务。在事务提交时，同步和刷新会像往常一样发生。Hibernate 会将所有排队的操作写到数据库。

也可以对分离实体实例的合并变更进行排队：

路径：/examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java

```
detachedItem.setName("New Name");
```

```
EntityManager em = JPA.createEntityManager();
```

```
Item mergedItem = em.merge(detachedItem);
```

```
tx.begin();
```

```
em.joinTransaction();
```

```
tx.commit();
```

```
em.close();
```

 刷新

当执行 `merge()` 操作时，Hibernate 会在自动提交模式中执行一个 SELECT。Hibernate 会在联结的事务提交前延迟执行 UPDATE 操作。

排队也适用于实体实例的移除和 DELETE 操作：

路径：/examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java

```
EntityManager em = JPA.createEntityManager();
```

```
Item item = em.find(Item.class, ITEM_ID);
```

```
em.remove(item);
```

```
tx.begin();
```

```
em.joinTransaction();
```

```
tx.commit();
```

```
em.close();
```

 刷新

因此，一个非同步的持久化上下文允许你从事务中解耦持久化操作。在本书后续内容我们探讨应用程序设计的时候，将必不可少地介绍 EntityManager 的这一特殊行为。对数据修改的排队功能，独立于事务(以及客户端/服务器请求)之外，是持久化上下文的主要功能。

Hibernate 的 MANUAL 刷新模式

Hibernate 提供了一个 `Session#setFlushMode()` 方法，具有额外的 `FlushMode.MANUAL`。即使是在联结的事务提交时，禁用持久化上下文的任何自动刷新都是更为方便的转换。使用此模式，必须显式调用 `flush()` 来与数据库保持同步。在 JPA 中，其思想是，“事务提交应

该总是写入任何未处理的变更”，这样一来，使用非同步模式就能将读取和写入分离开来。如果不同意这一点和/或不要自动提交的语句，则可以用 Session API 启用手动刷新。然后将常规的事务边界用于所有的工作单元，使用可重复的读取甚至是从 MVCC 数据库中隔离出来的快照，但仍然对持久化上下文中的变更进行排队以便用于后续的执行，并且在事务提交前手动刷新。

11.4 本章小结

- 学习了使用事务、并发、隔离和锁定。
- Hibernate 依赖数据库的并发控制机制，但在事务中提供了更好的隔离保障，这要归功于自动化版本控制以及持久化上下文缓存。
- 探讨了如何编程式设置事务边界以及处理异常。
- 探究了乐观并发控制以及显式的悲观锁。
- 介绍了如何使用自动提交模式以及事务之外的非同步持久化上下文，还有如何对修改进行排队。

在本章中，我们将探究用于 12.5 节中提到的导航的基础 ORM 问题的 Hibernate 解决方案。我们将介绍如何从数据库中检索数据以及如何优化这一加载。

Hibernate 提供了以下方法从数据库中检索数据并且放入内存中：

- 当已知一个实体实例的唯一标识符值时，通过标识符检索一个实体实例是最便利的方法：例如，`sessionManager.find(hibernate.class, 123)`。
- 可以导航实体图。从一个已经加载的实体实例开始，通过 `someItem.getSeller()`、`getAddress()`、`getCity()` 这样的属性访问器来联系访问关联的实例。当开始遍历一个集合时，所检索集合的元素也会按需加载。如果持久化上下文仍旧是开放的，那么 Hibernate 就会自动加载所需的节点。当调用访问器并且遍历集合时，加载哪些数据以及如何加载就是本章的重点了。
- 可以使用 Java 持久化查询语言 (Java Persistence Query Language, JPQL)，一个完全面向对象的查询语言，它基于 `select from Item where id = ?` 这样的字符串。
- CriteriaQuery 接口提供了一个类型安全和面向对象的方式来执行查询而无须字符串处理。
- 可以编写原生的 SQL 查询，调用在创建时，并让 Hibernate 负责将 JDBC 结果集映射到模型类型的实例。

在 Java 应用程序中，还要结合使用这些技术。我们不会在本章中非常详尽地探讨每一个检索方法。现在你应该已经熟悉了用于标识符检索的基础 Java 持久化 API。我们会让 JPQL 和 CriteriaQuery 示例尽可能地简单，并且不需要 SQL 查询映射功能。因为这些查询逻辑很复杂，我们将在后面的第 15~17 章探究它们。

抓取计划、策略和 配置文件

12

本章内容简介:

- 延迟加载和急加载
- 抓取计划、策略和配置文件
- 优化 SQL 执行

在本章中,我们将探究用于 1.2.5 节中提到过的导航的基础 ORM 问题的 Hibernate 解决方案。我们将介绍如何从数据库中检索数据以及如何优化这一加载。

Hibernate 提供了以下方式从数据库中抓取数据并且放入内存中:

- 当已知一个实体实例的唯一标识符值时,通过标识符检索一个实体实例是最便利的方法:例如, `entityManager.find(Item.class, 123)`。
- 可以导航实体图,从一个已经加载的实体实例开始,通过 `someItem.getSeller().getAddress().getCity()` 这样的属性访问器方法来访问关联的实例。当开始遍历一个集合时,所映射集合的元素也会按需加载。如果持久化上下文仍旧是开启的,那么 Hibernate 就会自动加载图形的节点。当调用访问器并且遍历集合时,加载哪些数据以及如何加载就是本章的重点了。
- 可以使用 Java 持久化查询语言(Java Persistence Query Language, JPQL),一个完全面向对象的查询语言,它基于 `select i from Item i where i.id = ?` 这样的字符串。
- `CriteriaQuery` 接口提供了一个类型安全和面向对象的方式来执行查询而无须字符串处理。
- 可以编写原生的 SQL 查询,调用存储过程,并且让 Hibernate 负责将 JDBC 结果集映射到域模型类的实例。

在 JPA 应用程序中,还要结合使用这些技术,我们不会在本章中非常详尽地探讨每一个检索方法。现在你应该已经熟悉了用于标识符检索的基础 Java 持久化 API。我们会让 JPQL 和 `CriteriaQuery` 示例尽可能地简单,并且不需要 SQL 查询-映射功能。因为这些查询选项很复杂,我们将在后面的第 15~17 章探究它们。

JPA 2 中主要的新功能

- 可以使用新的 `PersistenceUtil` 静态帮助器类手动检查一个实体或一个实体属性的初始化状态。

• 可以使用新的 EntityGraph API 创建标准的声明式抓取计划。

本章会介绍当导航域模型的图形以及 Hibernate 按需检索数据时，这些场景的背后会发生什么。在所有示例中，都会在注释中介绍当触发 SQL 执行的操作之后由 Hibernate 立即执行的 SQL。

Hibernate 加载什么取决于抓取计划：要定义应该被加载的对象网络的(子)图形。然后选取合适的抓取策略，定义应该如何加载数据。可以将所选择的计划和策略存储为一个抓取配置文件并且重复使用它。

定义抓取计划以及 Hibernate 应该加载哪些数据要依赖两个基础技术：延迟加载和急加载对象网络中的节点。

12.1 延迟加载和急加载

在某些时候，必须决定应该将哪些数据从数据库中加载到内存中。当执行 entityManager.find(Item.class, 123)时，哪些在内存中是可用的并且被加载到了持久化上下文中呢？如果转而使用 EntityManager#getReference()又会发生什么？

在域-模型映射中，要在关联和集合上使用 FetchType.LAZY 和 FetchType.EAGER 选项来定义全局默认的抓取计划。这个计划是用于所有涉及持久化域模型类的操作的默认设置。当通过标识符加载一个实体实例以及通过后续关联导航实体图并且遍历持久化集合时，它总是处于活动状态。

我们的推荐策略是将延迟的默认抓取计划用于所有实体和集合。如果使用 FetchType.LAZY 映射所有的关联和集合，那么 Hibernate 将只在你进行访问的时候加载数据。当导航域模型实例的图时，Hibernate 会按需一块一块地加载数据。然后在必要时基于每种情况重写此行为。

为实现延迟加载，Hibernate 借助被称为代理的运行时生成的实体占位符以及用于集合的智能包装器。

12.1.1 理解实体代理

思考一下 EntityManager API 的 getReference()方法。在 10.2.3 节中，已经首次介绍过这个操作以及它可以如何返回一个代理。我们现在进一步探究这一重要功能并且弄明白代理是如何运行的：

路径: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Item item = em.getReference(Item.class, ITEM_ID); ← 没有 SELECT
assertEquals(item.getId(), ITEM_ID); ← 调用标识符获取方法(没有字段访问)不会触发初始化
```

这段代码不会对数据库执行任何 SQL。Hibernate 所做的就是创建一个 Item 代理：它

看起来(以及听起来)像是真实的东西,但它仅仅是一个占位符。在持久化上下文中,即内存中,现在持久化状态中就提供了这个代理,如图 12-1 所示。

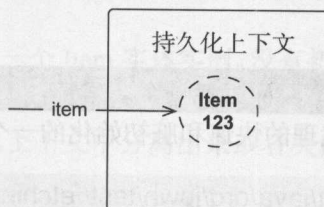


图 12-1 持久化上下文包含一个 Item 代理

该代理是一个运行时生成的 Item 子类的一个实例,带有它所表示的实体实例的标识符值。这就是 Hibernate(与 JPA 一致)为何需要实体类至少具有一个公共或受保护的无参构造函数(这个类可能也具有其他构造函数)的原因。该实体类及其方法不能被 final 修饰;否则, Hibernate 就无法生成一个代理。注意, JPA 规范并没有提及代理;延迟加载如何实现取决于 JPA 提供程序。

如果调用该代理上的任何并非“标识符获取方法”的方法就会触发代理的初始化并且访问数据库。如果调用 `item.getName()`, 那么将执行 SQL SELECT 以加载 Item。前一个示例调用了 `item.getId()` 而没有触发初始化, 是因为 `getId()` 是指定映射中的标识符获取方法; `getId()` 方法是由 `@Id` 注解的。如果 `@Id` 位于一个字段上, 则会调用 `getId()`, 就像调用其他任何方法一样, 这样就会初始化该代理!(要记住, 我们通常喜欢在字段上进行映射和访问, 因为这使得我们在设计访问器方法时有更多的自由;参阅 3.2.3 节。调用 `getId()` 而不初始化一个代理是否更重要, 这取决于你自己)。

使用代理时, 要当心比较类的方式。因为 Hibernate 会生成代理类, 它具有一个看起来有趣的名称, 并且它不等同于 `Item.class`:

路径: `/examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java`

```

assertNotEquals(item.getClass(), Item.class);
assertEquals(
    HibernateProxyHelper.getClassWithoutInitializingProxy(item),
    Item.class
);
  
```

类是在运行时生成的, 被命名为像 `Item_$$javassist_1` 这样的名称

如果真的必须得到代理表示的实际类型, 则可以使用 `HibernateProxyHelper`。JPA 提供了 `PersistenceUtil`, 可以使用它检查一个实体或其任何属性的初始化状态:

路径: `/examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java`

```

PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertFalse(persistenceUtil.isLoaded(item));
assertFalse(persistenceUtil.isLoaded(item, "seller"));
assertFalse(Hibernate.isInitialized(item));
// assertFalse(Hibernate.isInitialized(item.getSeller()));
  
```

会触发 item 的初始化

静态的 `isLoaded()` 方法也接受指定实体(代理)实例属性的名称, 以检查其初始化状态。Hibernate 用 `Hibernate.isInitialized()` 提供了一个可选的 API。不过, 如果调用 `item.getSeller()`, 则会首先初始化该 item 代理!

Hibernate 特性

Hibernate 还提供了用于代理的快速和脏初始化的一个实用工具方法:

路径: `/examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java`

```
Hibernate.initialize(item); // select * from ITEM where ID = ?
assertFalse(Hibernate.isInitialized(item.getSeller()));
Hibernate.initialize(item.getSeller()); // select * from USERS where ID = ?
```

确保已用
LAZY 重写了
@ManyToOne
的默认 EAGER

第一个调用会访问数据库并且加载 Item 数据, 用该商品的名称、价格等填充代理。

该 Item 的 seller 是一个用 `FetchType.LAZY` 映射的 `@ManyToOne` 关联, 因此 Hibernate 会在加载该 Item 时创建一个 User 代理。可以手动检查 seller 代理状态并加载它, 就像 Item 一样。记住, `@ManyToOne` 的 JPA 默认设置是 `FetchType.EAGER`! 你通常会希望重写它来得到延迟默认的抓取计划, 就像 7.3.1 节中首先介绍并且此处再次显示的那样:

路径: `/model/src/main/java/org/jpwh/model/fetching/proxy/Item.java`

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    public User getSeller() {
        return seller;
    }
    // ...
}
```

使用这样的延迟抓取计划, 可能遇到一个 `LazyInitializationException` 异常。思考以下代码:

路径: `/examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java`

```
Item item = em.find(Item.class, ITEM_ID); // ①加载 Item 实例
// select * from ITEM where ID = ?

em.detach(item); // ②分离数据
em.detach(item.getSeller());
// em.close(); // ③PersistenceUtil 帮助器

PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertTrue(persistenceUtil.isLoaded(item));
assertFalse(persistenceUtil.isLoaded(item, "seller"));
```

```
assertEquals(item.getSeller().getId(), USER_ID); ← ④ 调用获取方法
//assertNotNull(item.getSeller().getUsername()); ← 抛出一个异常
```

- ① 在持久化上下文中加载了一个 Item 实体实例。没有初始化它的卖家：它是一个 User 代理。
- ② 可以手动将数据从持久化上下文中分离出来或者关闭持久化上下文并且分离所有东西。
- ③ 静态的 PersistenceUtil 帮助器可以在无需持久化上下文的情况下运行。可以随时检查是否已经加载了你希望访问的数据。
- ④ 在分离状态中，可以调用 User 代理的标识符获取方法。但调用代理上像 getUsername() 这样的其他任何方法，都会抛出一个 LazyInitializationException 异常。只能在持久化上下文管理代理时按需加载数据，而不能在分离状态中加载。

一对一关联的延迟加载如何工作？

对于 Hibernate 新用户来说，有时会对一对一实体关联的延迟加载感到困惑。如果思考一下基于共享主键的一对一关联(参阅 8.1.1 节)，关联只能在其 optional=false 时才可以被代理。例如，Address 总是具有一个对 User 的引用。如果这个关联是可为空并且可选的，那么 Hibernate 就必须首先访问数据库来弄明白它是否应该应用一个代理或一个 null——并且延迟加载的目的是完全不访问数据库。可以通过字节码指令和拦截来启用可选的一对一关联的延迟加载，本章后续内容将探讨这一点。

Hibernate 代理的用处大于简单的延迟加载。例如，可以存储一个新的 Bid 而无须将任何数据加载进内存中：

```
Item item = em.getReference(Item.class, ITEM_ID);
User user = em.getReference(User.class, USER_ID);

Bid newBid = new Bid(new BigDecimal("99.00"));
newBid.setItem(item);
newBid.setBidder(user);

em.persist(newBid);
// insert into BID values (?, ?, ?, ...)
```

这段程序中没有 SQL SELECT，只有一个 INSERT

前两个调用会分别生成 Item 和 User 的代理。然后使用代理设置瞬时 Bid 的 item 和 bidder 关联属性。在刷新持久化上下文时，persist() 调用会将一个 SQL INSERT 排队，并且无须使用 SELECT 在 BID 表中创建新的行。所有的(外)键值都可用作 Item 和 User 代理的标识符值。

由 Hibernate 提供的运行时代理生成是用于透明延迟加载的绝佳选择。域模型类不必实现任何特殊(超)类型，而一些比较老的 ORM 解决方案会要求这一点。也不需要任何代码生成或字节码的后处理，这简化了构建过程。但应该当心一些可能的负面影响：

- 运行时代理并不完全透明的情况是多态关联，它们是用 instanceof 测试，这是 6.8.1 一节中介绍过的一个问题。

- 使用实体代理，就必须当心，不能在编写自定义 `equals()` 和 `hashCode()` 方法时直接访问字段，正如 10.3.2 节中所探讨过的一样。
- 代理仅能用于延迟加载实体关联。它们不能被用于延迟加载独立的基础属性或嵌入的组件，比如 `Item#description` 或 `User#homeAddress`。如果在这样一个属性上设置 `@Basic(fetch = FetchType.LAZY)` 提示，那么 Hibernate 会忽略它；在加载了所属实体实例时，值是被急加载的。尽管使用字节码指令和拦截是可行的，但我们认为这种优化没什么用处。如果正在使用(a)大量的可选/空列或者(b)包含由于系统物理限制而必须按需检索的大型值的列，那么在 SQL 中选择的单独列的级别进行优化就没有必要了。大型值最好转而使用定位器对象(Locator Object, LOB)来表示；根据定义它们提供了延迟加载。

代理会启用实体实例的延迟加载。对于持久化集合，Hibernate 提供了一个稍微不同的方法。

12.1.2 延迟持久化集合

要使用用于基础元素集合或可嵌入类型的 `@ElementCollection` 或者使用用于多值实体关联的 `@OneToMany` 和 `@ManyToMany` 来映射持久化集合。不同于 `@ManyToOne`，这些集合默认是延迟加载的。不必在映射上指定 `FetchType.LAZY` 选项。

当加载一个 `Item` 时，Hibernate 不会立即加载其 `images` 的延迟集合。该延迟 `bids` 一对多集合也是仅在被访问以及需要时才按需加载：

路径：/examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?

Set<Bid> bids = item.getBids();
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertFalse(persistenceUtil.isLoaded(item, "bids"));

assertTrue(Set.class.isAssignableFrom(bids.getClass()));
assertNotEquals(bids.getClass(), HashSet.class);
assertEquals(bids.getClass(),
    org.hibernate.collection.internal.PersistentSet.class);
```

集合未被初始化

← 它是一个 Set

← 它不是一个 HashSet

`find()` 操作会将 `Item` 实体实例加载到持久化上下文中，正如可以在图 12-2 中看到的那样。`Item` 实例具有对未初始化的 `User` 代理(seller)的引用。它还具有对未初始化的 `bids` 的 `Set` 以及未初始化的 `images` 的 `List` 的引用。

Hibernate 使用它的被称为集合包装器的专有特殊实现来实现集合的延迟加载(以及脏检查)。尽管 `bids` 看起来像一个 `Set`，但 Hibernate 还是会在你未注意时用一个 `org.hibernate.collection.internal.PersistentSet` 来替换该实现。它并非一个 `HashSet`，但它具有相同的行为。那就是为何使用域模型中的接口进行编程并且仅依赖 `Set` 而非 `HashSet` 如此重要的原因。列表和映射会以相同方式工作。

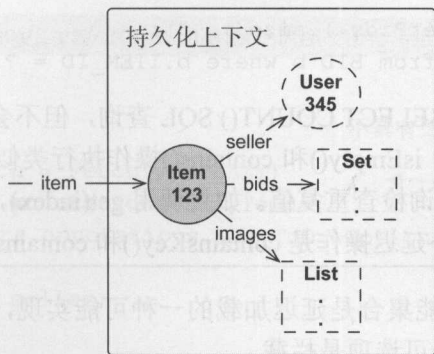


图 12-2 表示所加载图形边界的代理和集合包装器

这些特殊的集合可以检测到访问它们的时间并且在该时间加载其数据。只要开始遍历 bids，就会加载集合以及所有对该商品做出的出价：

路径：/examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Bid firstBid = bids.iterator().next();
// select * from BID where ITEM_ID = ?

// Alternative: Hibernate.initialize(bids);
```

或者，就像为实体代理所做的那样，可以调用 Hibernate.initialize() 这个静态工具方法来加载一个集合。它将被完全加载；例如，无法“只加载前两个出价”。为此，必须编写一个查询。

Hibernate 特性

为方便起见，不必编写许多琐碎的查询，Hibernate 为集合映射提供了一个专用设置：

路径：/model/src/main/java/org/jpwh/model/fetching/proxy/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.LazyCollection(
        org.hibernate.annotations.LazyCollectionOption.EXTRA
    )
    public Set<Bid> getBids() {
        return bids;
    }

    // ...
}
```

使用 LazyCollectionOption.EXTRA，集合就可以支持不会触发初始化的操作。例如，可以查询该集合的大小：

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?
```

```
assertEquals(item.getBids().size(), 3);
// select count(b) from BID b where b.ITEM_ID = ?
```

size()操作会触发一个 SELECT COUNT() SQL 查询, 但不会将 bids 加载到内存中。在所有额外延迟集合上, 都为 isEmpty()和 contains()操作执行类似查询。当调用 add()时, 额外延迟 Set 会用一个简单查询检查重复值。如果调用 get(index), 则额外延迟 List 只会加载一个元素。对于 Map, 额外延迟操作是 containsKey()和 containsValue()。

Hibernate 的代理和智能集合是延迟加载的一种可能实现, 具有功能与开销的良好平衡。我们之前提到过的一个可选项是拦截。

Hibernate 特性

12.1.3 使用拦截进行延迟加载

使用延迟加载的基本问题是, JPA 提供程序必须清楚何时加载 Item 的 seller 或者 bids 的集合。相较于运行时生成的代理和智能集合, 其他许多 JPA 提供程序都主要依赖方法调用的拦截。例如, 在调用 someItem.getSeller()时, JPA 提供程序会拦截这个调用并且加载表示 seller 的 User 实例。

这个方法需要 Item 类中有特殊的代码来实现拦截: 必须包装 getSeller()方法或者 seller 字段。因为不希望手动编写这部分代码, 所以通常要在编译域模型类之后运行字节码增强器(与 JPA 提供程序绑定在一起的)。这个增强器会将必要的拦截代码注入到编译过的类中, 以便在字节码级别处理字段和方法。

我们用一些示例探讨基于拦截的延迟加载。首先, 你可能希望禁用 Hibernate 的代理生成:

路径: /model/src/main/java/org/jpwh/model/fetching/interception/User.java

```
@Entity
@org.hibernate.annotations.Proxy(lazy = false)
public class User {
    // ...
}
```

Hibernate 现在将不再为 User 实体生成一个代理。如果调用 entityManager.getReference(User.class, USER_ID), 则会执行一个 SELECT, 就像为 find()所做的那样:

路径: /examples/src/test/java/org/jpwh/test/fetching/LazyInterception.java

```
User user = em.getReference(User.class, USER_ID);
// select * from USERS where ID = ?
assertTrue(Hibernate.isInitialized(user));
```

代理被禁用了。getReference()将返回一个初始化的实例

对于关联目标是 User 的实体, 比如 Item 的 seller, FetchType.LAZY 提示没有任何效果:

路径: /model/src/main/java/org/jpwh/model/fetching/interception/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    @org.hibernate.annotations.LazyToOne(
        org.hibernate.annotations.LazyToOneOption.NO_PROXY
    )
    protected User seller;
    // ...
}
```

不具有效果——没有 User 代理

需要字节码增强

相反，专有的 `LazyToOneOption.NO_PROXY` 设置会告知 Hibernate，字节码增强器必须为 `seller` 属性添加拦截代码。不使用此选项的话，或者如果不运行字节码增强器的话，这个关联就会是急加载并且加载 `Item` 时会立即填充该字段，因为 `User` 实体的代理已经被禁用了。

如果运行该字节码增强器，Hibernate 就会拦截 `seller` 字段的访问并且在触及该字段时触发加载：

路径: /examples/src/test/java/org/jpwh/test/fetching/LazyInterception.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?

assertEquals(item.getSeller().getId(), USER_ID);
// select * from USERS where ID = ?
```

即使 item.getSeller()也会触发 SELECT

这比代理具有更少的延迟。记住，可以在代理上调用 `User#getId()` 而不会初始化实例，正如之前几节中所阐释的一样。使用拦截，所有对 `seller` 字段的访问以及对 `getSeller()` 的调用都会触发初始化。

对于延迟实体关联，代理通常是比拦截更好的选择。拦截的一个更常见用例是可能具有大型值的基本类型的属性，比如 `String` 或 `byte[]`。我们可能会认为 LOB 应该更适用于大型字符串或二进制数据，但你可能不希望在域模型中使用 `java.sql.Blob` 或 `java.sql.Clob` 类型。使用拦截和字节码增强，就可以按需加载一个简单的 `String` 或 `byte[]` 字段：

路径: /model/src/main/java/org/jpwh/model/fetching/interception/Item.java

```
@Entity
public class Item {

    @Basic(fetch = FetchType.LAZY)
    protected String description;
    // ...
}
```

如果在编译过的类上运行字节码增强器，就将延迟加载 `Item#description`。如果不运行该字节码增强器——例如，在开发期间——则 `String` 将与 `Item` 实例一起被加载。

如果借助拦截，则要当心 Hibernate 将加载实体或可嵌入类的所有延迟字段，即便只有一个字段必须被加载时也会如此：

路径：/examples/src/test/java/org/jpwh/test/fetching/LazyInterception.java

```
Item item = em.find(Item.class, ITEM_ID);
// select NAME, AUCTIONEND, ... from ITEM where ID = ? 访问一个延迟属性会
assertTrue(item.getDescription().length() > 0); ←——加载所有延迟属性
// select DESCRIPTION from ITEM where ID = ? (description、seller 等)
// select * from USERS where ID = ?
```

当 Hibernate 加载 Item 的 description 时，它也会立即加载 seller 和任何其他拦截到的字段。在编写本书时，Hibernate 中还没有抓取分组：要么全部抓取，要么什么都不抓取。

拦截的缺点是每次构建域模型类时运行一个字节码增强器的开销以及等待指令完成的时间。如果应用程序的行为不依赖一个商品的描述加载状态，那么在开发期间可以决定跳过该指令。然后，在构建测试和生产包时，可以执行该增强器。

新的 Hibernate 5 字节码增强器

遗憾的是，我们不能确保此处介绍的拦截示例能够用于最新的 Hibernate 5 版本。Hibernate 5 的字节码增强器已经被重写了，并且现在支持的要比用于延迟加载的拦截更多：字节码增强器可以将代码注入到域模型类中以便加快脏检查并且自动管理双向实体关联。不过，在撰写本书时，我们无法获得这个全新的增强器来进行研究，且开发仍在继续。我们建议你阅读当前的 Hibernate 文档以便获得更多与该增强器功能有关的信息，并且使用 Maven 或 Gradle 插件在项目中配置它。

我们将是否希望将拦截用于延迟加载的决定留给你自己——根据我们的经验，很少有好的用例。注意，我们在探讨拦截时还没有讨论到集合包装器：尽管可以为集合字段启用拦截，但 Hibernate 仍旧会使用其智能集合包装器。原因在于，不同于实体代理，这些集合包装器用于延迟加载以外的其他目的。例如，在脏检查时，Hibernate 会借助它们来跟踪集合元素的添加和移除。你不能在映射中禁用该集合包装器；它们总是开启的（当然，你永远不必映射一个持久化集合；它们是一个功能，而非一个需求。另一方面，仅能使用字段拦截来延迟加载持久化数组——不能像集合那样包装它们。

现在你已经看到了 Hibernate 中所有用于延迟加载的可用选项。接下来看看按需加载的对立面：数据的急抓取。

12.1.4 关联和集合的急加载

我们已经推荐了一个延迟默认抓取计划，在所有的关联和集合映射上使用 FetchType.LAZY。有时候，你会希望使用相反的方式：指定某特定实体关联或集合应该总是被加载。你希望确保这个数据在内存中可用，而不需要额外的数据库访问。

更重要的是，举个例子来说，你希望确保可以在 Item 实例处于分离状态时访问 Item

的 seller。当持久化上下文被关闭时，延迟加载就不再可用。如果 seller 是一个未初始化的代理，那么在分离状态中访问它时，就会得到一个 LazyInitializationException 异常。为了让数据在分离状态中可用，需要在持久化上下文仍然开启时手动加载它，或者，如果希望它总是被加载，则要将你的抓取计划修改为急抓取而非延迟抓取。

我们假设你总是需要加载 seller 和 Item 的 bids:

路径: /model/src/main/java/org/jpwh/model/fetching/eagerjoin/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER) ←—— 默认设置
    protected User seller;

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER) ←—— 不建议的
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

不同于作为 JPA 提供程序可以忽略的提示的 FetchType.LAZY，FetchType.EAGER 是一个强制要求。提供程序必须确保数据可以被加载并且在分离状态中可用；它不能忽略该设置。

思考一下集合映射：“无论何时将一个商品加载到内存中，也都要立即加载该商品的出价吗”这个说法真的合适吗？即便你只希望显示该商品的名称或在拍卖结束时找到它，也会将所有的出价加载到内存中。总是在映射中使用 FetchType.EAGER 作为默认抓取计划的急加载集合，通常并非一个好的策略。如果急加载几个集合，还会看到笛卡尔积问题的出现，本章稍后将探讨这一点。最好对集合使用默认的 FetchType.LAZY。

如果现在 find() 一个 Item(或强制执行 Item 代理的初始化)，那么 seller 和所有的 bids 都会作为持久化实例被加载到你的持久化上下文中：

路径: /examples/src/test/java/org/jpwh/test/fetching/EagerJoin.java

```
Item item = em.find(Item.class, ITEM_ID);
// select i.*, u.*, b.*
// from ITEM i
// left outer join USERS u on u.ID = i.SELLER_ID
// left outer join BID b on b.ITEM_ID = i.ID
// where i.ID = ?

em.detach(item);

assertEquals(item.getBids().size(), 3);
assertNotNull(item.getBids().iterator().next().getAmount());

assertEquals(item.getSeller().getUsername(), "johndoe");
```

抓取完成：不再有延迟加载

在分离状态中，bids 是可用的.....

.....且 seller 也是可用的

对于 `find()`，Hibernate 会执行单个 SQL `SELECT` 并且 `JOIN` 三个表来检索数据。可以在图 12-3 中看到该持久化上下文的内容。注意是如何表示所加载图形的边界的：还没有加载 `images` 的集合，并且每个 `Bid` 都有一个对未初始化 `User` 代理的引用，即 `bidder`。如果现在分离该 `Item`，就可以访问所加载的 `seller` 和 `bids`，而不会引发 `LazyInitializationException` 异常。如果尝试访问 `images` 或其中一个 `bidder` 代理，就会得到一个异常。

在后面的示例中，我们假设你的域模型具有一个延迟默认抓取计划，Hibernate 仅会加载显式请求的数据以及访问的关联和集合。

接下来，我们要探讨当通过标识找到一个实体实例以及当导航该网络时应该如何使用所映射关联和集合的指针来加载数据。我们关注的是执行哪些 SQL 以及找出理想的抓取策略。

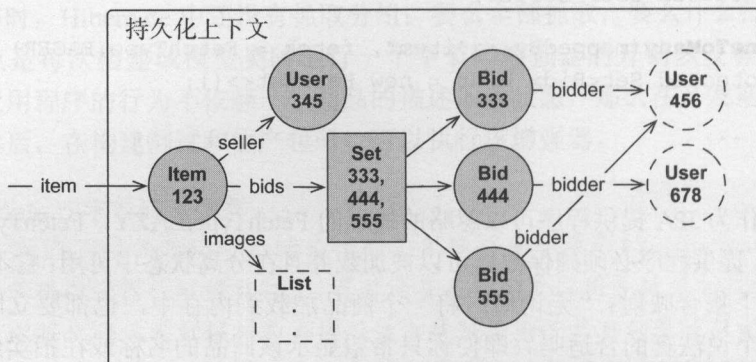


图 12-3 加载了 `Item` 的 `seller` 和 `bids`

12.2 选择一个抓取策略

Hibernate 会执行 SQL `SELECT` 语句将数据加载到内存中。如果加载一个实体实例，则会执行一个或多个 `SELECT`，这取决于涉及的表数量以及所应用的抓取策略。你的目标是 minimized SQL 语句的数量并且简化 SQL 语句，以便查询尽可能提高效率。

思考一下本章早前推荐的抓取计划：每一个关联和集合都应该按需被延迟加载。这一默认抓取计划很可能造成过多的 SQL 语句，每个语句都仅加载一小部分数据。这将导致 $n+1$ 次查询问题，我们首先探讨这个问题。使用急加载这一可选抓取计划，将产生较少的 SQL 语句，因为每个 SQL 查询都会将较大块的数据加载到内存中。然后你可能会看到笛卡尔积问题，因为 SQL 结果集变得过大。

需要在这两个极端之间找到平衡：用于应用程序中每个程序和用例的理想抓取策略。就像抓取计划一样，可以在映射中设置一个全局抓取策略：总是生效的默认设置。然后，对于某特定程序，可以用自定义 JPQL、CriteriaQuery 或 SQL 查询重写此默认抓取策略。

首先，我们来探讨你看到的基础问题，从 $n+1$ 查询问题开始。

12.2.1 $n+1$ 查询问题

用一些示例代码就可以很容易地理解这个问题。我们假设你映射了一个延迟抓取计

划，这样就会按需加载所有东西。以下示例代码会检查每个 Item 的 seller 是否具有一个 username:

路径: /examples/src/test/java/org/jpwh/test/fetching/NPlusOneSelects.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
    // select * from USERS where ID = ?
}
```

必须使用额外的
SELECT 加载每
一个 seller

你看到了加载 Item 实体实例的一个 SQL SELECT。然后，当遍历所有 items 时，检索每个 User 都需要一个额外的 SELECT。这相当于用于 Item 的一个查询加上 n 个查询，n 取决于你有多少商品以及特定 User 是否在拍卖多个 Item。显然，如果知道你要访问每个 Item 的 seller 的话，就会认为这是一个非常低效的策略。

使用延迟加载的集合你会看到相同的问题。以下示例会检查每个 Item 是否具有一些 bids:

路径: /examples/src/test/java/org/jpwh/test/fetching/NPlusOneSelects.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID = ?
}
```

每个 bids 集合都必须使用一个
额外的 SELECT 加载

同样，如果知道你要访问每个 bids 集合的话，就会认为一次只加载一个商品是低效的。如果有 100 个商品，就要执行 101 个 SQL 查询！

基于你目前所了解的情况，你可能会倾向于在映射中修改默认的抓取计划并且在你的 seller 或 bids 关联上放置一个 FetchType.EAGER。但这样做会导致我们要探讨的下一个主题：笛卡尔积问题。

12.2.2 笛卡尔积问题

如果查看域和数据模型并且认为，“每次我需要一个 Item 时，我还需要该 Item 的 seller”，那么就可以使用 FetchType.EAGER 而非延迟抓取计划来映射该关联。你希望确保无论何时加载一个 Item，seller 都会被立即加载——你希望数据在分离 Item 和关闭持久化上下文时可用：

路径: /model/src/main/java/org/jpwh/model/fetching/cartesianproduct/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)
    protected User seller;
```

```
// ...
}
```

为实现急抓取计划，Hibernate 使用了一个 SQL JOIN 操作在一个 SELECT 中加载 Item 和 User 实例：

```
item = em.find(Item.class, ITEM_ID);
// select i.*, u.*
// from ITEM i
// left outer join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

该结果集包含一行组合 USERS 表数据的来自 ITEM 表的数据，如图 12-4 所示。

i.ID	i.NAME	i.SELLER_ID	...	u.ID	u.USERNAME	...
1	One	2	...	2	johndoe	...

图 12-4 Hibernate 联结两个表来急抓取关联行

将使用默认 JOIN 策略的急抓取用于@ManyToOne 和@OneToOne 关联没什么问题。可以使用一个 SQL 查询和多个 JOIN 急加载一个 Item、其 seller、该 User 的 Address 以及他们居住的 City 等。即便你使用 FetchType.EAGER 映射所有这些关联，结果集也只有一行。现在，Hibernate 必须在某个时刻停止继续你的 FetchType.EAGER 计划。所联结的表的数量取决于全局的 hibernate.max_fetch_depth 配置属性。默认情况下，不会设置任何限制。合理值很小，通常介于 1 到 5 之间。甚至可以通过将该属性设置为 0 来禁用@ManyToOne 和@OneToOne 关联的 JOIN 抓取。如果 Hibernate 达到了该限制，那么它仍将根据你的抓取计划急加载数据，但会使用额外的 SELECT 语句(注意有些数据库方言可能会预设这个属性：例如，MySQLDialect 会将它设置为 2)。

另一方面，使用 JOINS 的急加载集合会导致严重的性能问题。如果也为 bids 和 images 集合切换到 FetchType.EAGER，就会碰到笛卡尔积问题。

这个问题会在用一个 SQL 查询和一个 JOIN 操作急加载两个集合时出现。首先，创建这样一个抓取计划，然后看看该 SQL 问题：

路径：/model/src/main/java/org/jpwh/model/fetching/cartesianproduct/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
    protected Set<Bid> bids = new HashSet<>();

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    protected Set<String> images = new HashSet<String>();

    // ...
}
```

这两个集合是@OneToMany、@ManyToMany 还是@ElementCollection 并没有什么关系。使用 SQL JOIN 操作符一次性急抓取多个集合就是根本问题，无论集合内容是什么。如果加载一个 Item，那么 Hibernate 就会执行有问题的 SQL 语句：

路径：/examples/src/test/java/org/jpwh/test/fetching/CartesianProduct.java

```
Item item = em.find(Item.class, ITEM_ID);
// select i.*, b.*, img.*
//   from ITEM i
//  left outer join BID b on b.ITEM_ID = i.ID
//  left outer join IMAGE img on img.ITEM_ID = i.ID
// where i.ID = ?

em.detach(item);

assertEquals(item.getImages().size(), 3);
assertEquals(item.getBids().size(), 3);
```

正如可以看到的，Hibernate 会服从你的急抓取计划，并且可以访问分离状态中的 bids 和 images 集合。问题在于，使用产生一个乘积的 SQL JOIN，这些集合是如何被加载的。看看图 12-5 中的结果集。

i.ID	i.NAME	...	b.ID	b.AMOUNT	img.FILENAME
1	One	...	1	99.00	foo.jpg
1	One	...	1	99.00	bar.jpg
1	One	...	1	99.00	baz.jpg
1	One	...	2	100.00	foo.jpg
1	One	...	2	100.00	bar.jpg
1	One	...	2	100.00	baz.jp
1	One	...	3	101.00	foo.jpg
1	One	...	3	101.00	bar.jpg
1	One	...	3	101.00	baz.jpg

图 12-5 一个乘积是两个与多行联结的结果

这个结果集包含许多多余的数据项，其中只有阴影的单元格才与 Hibernate 相关。该 Item 具有 3 个 bids 和 3 个 images。乘积的大小取决于你正在检索的集合的大小：3 乘以 3 总计为 9 行。现在思考有一个具有 50 个 bids 和 5 个 images 的 Item 的情况——你会看到可能具有 250 行的一个结果集！在使用 JPQL 或 CriteriaQuery 编写你自己的查询时你甚至会创建更大的 SQL 乘积：想象一下你在加载 500 个 items 并且使用多个 JOIN 急抓取几十个 bids 和 images 时会发生什么。

数据库服务器上需要大量的处理时间和内存来创建这样的结果，这些结果还必须跨网络传输。如果寄希望于 JDBC 驱动设法在传输时压缩该数据，那你可能对数据库供应商的期望过高了。Hibernate 会在将结果集封送到持久化实例和集合中时立即移除所有的重复项；图 12-5 中非阴影部分的单元格中的信息将被忽略。显然，无法在 SQL 级别移除这些重复项；SQL DISTINCT 操作符在这里起不到什么作用。

相对于具有一个异常大的结果的 SQL 查询，同时用三个独立的查询检索一个实体实例和两个集合会更快一些。接下来，我们要专注于此类优化以及如何找出并且实现最佳的抓取策略。我们还是从默认延迟抓取计划开始并且首先尝试解决 $n+1$ 查询问题。

Hibernate 特性

12.2.3 批量预抓取数据

如果 Hibernate 仅按需抓取每个实体关联和集合，那么可能就需要许多额外的 SQL SELECT 语句来完成某特定过程。像之前一样，思考一个检查每个 Item 的 seller 是否具有一个 username 的例程。使用延迟加载，就需要一个 SELECT 得到所有的 Item 实例以及更多的 n 个 SELECT 来初始化每个 Item 的 seller 代理。

Hibernate 提供了几个可以预抓取数据的算法。我们要探讨的第一个算法是批量预抓取，它会如下所示地工作：如果 Hibernate 必须初始化一个 User 代理，那么就使用相同 SELECT 初始化几个 User 代理。换句话说，如果已经知道持久化上下文中有几个 Item 实例并且它们都具有一个应用到其 seller 关联的代理，那么就可以初始化几个代理，而不是在与数据库交互时只初始化一个代理。

我们看看这是如何工作的。首先，使用一个专有的 Hibernate 注解启用 User 实例的批量抓取：

路径：/model/src/main/java/org/jpwh/model/fetching/batch/User.java

```
@Entity
@org.hibernate.annotations.BatchSize(size = 10)
@Table(name = "USERS")
public class User {
    // ...
}
```

此设置会告知 Hibernate，在必须加载一个 User 代理时它可以加载至 10 个，所有的代理都使用相同 SELECT 来加载。批量抓取通常被称为盲猜优化，因为你不知道某特定持久化上下文中会有多少个未初始化的 User 代理。你不能确定 10 是否是一个理想值——它只是一个猜测而已。你清楚相较于 $n+1$ 个 SQL 查询，你现在会看到 $n+1/10$ 个查询，已经显著减少了。合理的值通常很小，因为你也不希望将过多的数据加载到内存中，尤其是在你不确定是否需要它时。

下面就是优化过的过程，它会检查每个 seller 的 username：

路径：/examples/src/test/java/org/jpwh/test/fetching/Batch.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
    // select * from USERS where ID in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
}
```

注意 Hibernate 在你遍历 items 时执行的 SQL 查询。当首次调用 `item.getSeller().getUserName()` 时, Hibernate 必须初始化第一个 User 代理。相较于仅从 USERS 表中加载单个行, Hibernate 会检索多个行, 并且加载最多 10 个 User 实例。一旦访问第 11 个 seller, 就会在一个批次中加载另外 10 个, 以此类推, 直到持久化上下文不再包含未初始化的 User 代理为止。

常见问题解答: 什么是真正的批量抓取算法?

我们对于批量加载的阐释是有一些简化的, 实际上你可能会看到稍微不同的算法。举例来说, 想象一下 32 这一批量大小。在启动时, Hibernate 会内部创建几个批量加载器。每个加载器都知道它可以初始化多少个代理: 32、16、10、9、8、7、.....、1。其目标是加载器的创建最小化内存消耗, 并且足够多的加载器能生成每一个可能的批量抓取。很明显, 另一个目标就是最小化 SQL 查询的数量。

为初始化 31 个代理, Hibernate 要执行 3 个批量(你可能预期的是 1 个, 因为 $32 > 31$)。所应用的批量加载器是 16、10 和 5, 是由 Hibernate 自动选取的。可在持久化单元配置中使用属性 `hibernate.batch_fetch_style` 自定义此批量抓取算法。其默认设置是 LEGACY, 它会在启动时构建并选择几个批量加载器。其他的选项是 PADDED 和 DYNAMIC, 使用 PADDED, Hibernate 只会在启动时构建一个批量加载器 SQL 查询, 它的 IN 子句中具有用于 32 个参数的占位符, 然后在必须加载小于 32 个代理时重复绑定标识符。使用 DYNAMIC, 当 Hibernate 知道要初始化的代理数量时, 它就会在运行时动态地构建批量 SQL 语句。

批量抓取也可用于集合:

路径: `/model/src/main/java/org/jpwh/model/fetching/batch/Item.java`

@Entity

```
public class Item {
```

```
    @OneToMany(mappedBy = "item")
```

```
    @org.hibernate.annotations.BatchSize(size = 5)
```

```
    protected Set<Bid> bids = new HashSet<>();
```

```
    // ...
```

```
}
```

如果现在强制一个 bids 集合的初始化, 达到更多的 5 个 Item#bids 集合, 如果在当前持久化上下文中还未初始化它们, 则它们会立即被加载:

路径: `/examples/src/test/java/org/jpwh/test/fetching/Batch.java`

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM
```

```
for (Item item : items) {
```

```
    assertTrue(item.getBids().size() > 0);
```

```
    // select * from BID where ITEM_ID in (?, ?, ?, ?, ?)
```

```
}
```

当在遍历期间首次调用 `item.getBids().size()` 时, 整批 Bid 集合都会被预加载以用于其他

Item 实例。

批量抓取是简单的，并且通常智能优化能够显著降低 SQL 语句的数量，否则初始化所有的代理和集合就需要大量 SQL 语句。尽管最终可能会预抓取你不需要的数据，并且消耗更多的内存，但数据库交互的减少也会产生很大的差异。内存很便宜，但扩展数据库服务器就并非如此了。

另一个并非盲猜的预抓取算法会使用子查询在单个语句中初始化多个集合。

Hibernate 特性

12.2.4 使用子查询预抓取集合

用于加载几个 Item 实例的所有 bids 的更好的一个策略是使用一个子查询进行预抓取。要启用此优化，需要将一个 Hibernate 注解添加到你的集合映射：

路径：/model/src/main/java/org/jpwh/model/fetching/subselect/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SUBSELECT
    )
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Hibernate 现在可以在你强制一个 bids 集合的初始化时立即为所有加载的 Item 实例初始化所有的 bids 集合：

路径：/examples/src/test/java/org/jpwh/test/fetching/Subselect.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID in (
    // select ID from ITEM
    // )
}
```

Hibernate 会记住用于加载 items 的原始查询。然后它会在子查询中嵌入这个初始查询（稍经修改），以便为每个 Item 检索 bids 的集合。

使用子查询的预抓取是一个很强大的优化，但在撰写本书时，它仅可用于延迟集合，无法用于实体代理。还要注意，作为一个子查询返回的原始查询仅会由 Hibernate 为特定持久化上下文记住。如果分离一个 Item 实例而没有初始化 bids 的集合，然后将其与一个新

的持久化上下文合并并且开始遍历该集合，则不会出现其他集合的预抓取。

如果在映射中坚持使用一个全局延迟抓取计划，那么批量和子查询预抓取就会降低特定过程需要的查询数量，以帮助缓解 $n+1$ 查询问题。如果相反，你的全局抓取计划具有急加载关联和集合，就必须避免笛卡尔积问题——例如，通过将一个 JOIN 查询分解为几个 SELECT 来避免。

Hibernate 特性

12.2.5 使用多个 SELECT 进行急抓取

当尝试用一个 SQL 查询和多个 JOIN 抓取几个集合时，就会碰到笛卡尔积问题，就像之前阐释过的那样。相较于一个 JOIN 操作，可以告知 Hibernate 用几个额外的 SELECT 查询急加载数据，并因而避免大的结果以及具有重复项的 SQL 乘积：

路径：/model/src/main/java/org/jpwh/model/fetching/eagerselect/Item.java

@Entity

```
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT
    )
    protected User seller;

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT
    )
    protected Set<Bid> bids = new HashSet<>();
    // ...
}
```

默认为
JOIN

现在，当加载一个 Item 时，也必须加载 seller 和 bids：

路径：/examples/src/test/java/org/jpwh/test/fetching/EagerSelect.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?
// select * from USERS where ID = ?
// select * from BID where ITEM_ID = ?

em.detach(item);
assertEquals(item.getBids().size(), 3);
assertNotNull(item.getBids().iterator().next().getAmount());
assertEquals(item.getSeller().getUsername(), "johndoe");
```

Hibernate 会使用一个 SELECT 从 ITEM 表中加载一行。然后它会立即执行两个

SELECT: 一个从 USERS 表中加载一行(seller), 另一个从 BID 表中加载几行(bids)。

额外的 SELECT 查询不会被延迟执行; find()方法会生成几个 SQL 查询。可以看到 Hibernate 如何遵循急抓取计划: 所有数据在分离状态下都是可用的。

所有这些设置依旧是全局的; 它们总是生效的。风险在于, 为应用程序中一个有问题的情况调整一个设置可能会对其他一些过程产生负面影响。维持这一平衡可能会很难, 因此我们建议, 将每个实体关联和集合映射为 FetchType.LAZY, 就像之前提到过的那样。

一种更好的方法是, 仅在需要时为特定过程动态使用急抓取和 JOIN 操作。

12.2.6 动态急抓取

就像前面几节一样, 我们假设你必须检查每个 Item#seller 的 username。使用一个延迟全局抓取计划, 加载这个过程所需的数据并且在一个查询中应用动态急抓取策略:

路径: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
List<Item> items =
    em.createQuery("select i from Item i join fetch i.seller")
      .getResultList();
// select i.*, u.*
// from ITEM i
// inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?

em.close();
```

← 分离所有东西

```
for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
}
```

这个 JPQL 查询中的重要关键字是 join fetch, 告知 Hibernate 使用一个 SQL JOIN (实际上是一个 INNER JOIN)在相同查询中检索每个 Item 的 seller。可使用 CriteriaQuery API 而非 JPQL 字符串来表示相同查询:

路径: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);
i.fetch("seller");
criteria.select(i);

List<Item> items = em.createQuery(criteria).getResultList();

em.close();
```

← 分离所有东西

```
for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
}
```

动态急联结抓取也适用于集合。此处要加载每个 Item 的所有 bids:

路径: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
List<Item> items =
    em.createQuery("select i from Item i left join fetch i.bids")
      .getResultList();
// select i.*, b.*
// from ITEM i
// left outer join BID b on b.ITEM_ID = i.ID
// where i.ID = ?

em.close();
```

← 分离所有东西

```
for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
}
```

现在也同样使用 CriteriaQuery API:

路径: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();

Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
criteria.select(i);

List<Item> items = em.createQuery(criteria).getResultList();

em.close();
```

← 分离所有东西

```
for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
}
```

注意, 对于集合抓取, LEFT OUTER JOIN 是必要的, 因为就算没有 bids, 你也希望得到 ITEM 表的行。我们在本书后续的第 15 章中将更详细地介绍使用 JPQL 和 CriteriaQuery 进行抓取。到时候你将看到更多内联结、外联结、左联结和右联结的示例, 所以目前不要太过担心这些细节。

如果希望动态重写域模型的全局抓取计划, 手动编写查询就并非唯一可用的选项。可以声明式编写抓取配置文件。

12.3 使用抓取配置文件

抓取配置文件为查询语言和 API 中的抓取选项提供了补充。它们允许在 XML 或者注解元数据中维护配置文件定义。Hibernate 的早期版本没有对特殊抓取配置文件的支持, 但现在的 Hibernate 支持以下内容:

- 抓取配置文件——一个基于使用@org.hibernate.annotations.FetchProfile 的配置文件声明以及使用 Session#enableFetchProfile()执行的专用 API。此简单机制当前支持选择性地重写延迟映射的实体关联和集合，以便为特定工作单元启用一个 JOIN 急抓取策略。
- 实体图——在 JPA 2.1 中明确规定的，可以使用@EntityGraph 注解声明一个实体属性和关联的图形。这一抓取计划，或一组计划，可以在执行 EntityManager#find() 或查询(JPQL, 标准)时作为一个提示来启用。所提供的图形会控制应该加载什么；遗憾的是它并不控制应该如何加载。

可以认为此处还有改进的空间，我们期望将来的 Hibernate 和 JPA 版本提供一个统一且更强大的 API。

不要忘记可以外部化 JPQL 和 SQL 语句并且将它们移动到元数据中(参阅 14.4 节)。JPQL 查询是声明式地抓取配置文件；你所缺少的就是在相同的基查询上轻易叠加不同计划的能力。我们已经看到过一些具有字符串处理的创造性解决方案，但却很少被使用。另一方面，使用标准查询，就已经具有了 Java 可用的完整能力来组织你的查询构建代码。然后实体图的值就能够跨任何类型的查询重用抓取计划。

我们首先来探讨 Hibernate 抓取配置文件，以及如何为特定工作单元重写一个全局延迟抓取计划。

Hibernate 特性

12.3.1 声明 Hibernate 抓取配置文件

Hibernate 抓取配置文件是全局元数据：它们是为整个持久化单元声明的。尽管可以在一个类上放置@FetchProfile 注解，但我们更倾向于将它用作 package-info.java 中的包级别元数据：

路径：/model/src/main/java/org/jpwh/model/fetching/profile/package-info.java

```
@org.hibernate.annotations.FetchProfiles({
    @FetchProfile(name = Item.PROFILE_JOIN_SELLER, ← ❶ 配置文件名称
        fetchOverrides = @FetchProfile.FetchOverride(← ❷ 重写
            entity = Item.class,
            association = "seller",
            mode = FetchMode.JOIN ← ❸ JOIN 模式
        ),
    @FetchProfile(name = Item.PROFILE_JOIN_BIDS,
        fetchOverrides = @FetchProfile.FetchOverride(
            entity = Item.class,
            association = "bids",
            mode = FetchMode.JOIN
        )
})
```

❶ 每个配置文件都有一个名称。这是隔离在一个常量中的简单字符串。

② 配置文件中的每个重写都要命名一个实体关联或集合。

③ 在编写本书时唯一支持的模式就是 JOIN。

现在可为工作单元启用配置文件：

路径：/examples/src/test/java/org/jpwh/test/fetching/Profile.java

```
Item item = em.find(Item.class, ITEM_ID);           ← ①检索实例

em.clear();
em.unwrap(Session.class).enableFetchProfile(Item.PROFILE_JOIN_SELLER); ← ②启用配置文件
item = em.find(Item.class, ITEM_ID);

em.clear();
em.unwrap(Session.class).enableFetchProfile(Item.PROFILE_JOIN_BIDS); ← ③叠加第二个配置文件
item = em.find(Item.class, ITEM_ID);
```

① Item#seller 是被延迟映射的，所以默认的抓取计划仅会检索该 Item 实例。

② 需要 Hibernate API 启用一个配置文件。然后在该工作单元中它可用于任何操作。无论何时用此 EntityManager 加载一个 Item，都会在相同 SQL 语句中使用一个联结来抓取该 Item#seller。

③ 可以在相同的工作单元上叠加另一个配置文件。现在，无论何时加载一个 Item，都会在相同 SQL 语句中使用一个联结抓取 Item#seller 和 Item#bids 集合。

尽管很基础，但对于较小或较简单应用程序中的抓取优化来说，Hibernate 抓取配置文件也会是一种轻松的解决方案。使用 JPA 2.1，实体图的引入就可以用标准方式启用类似功能。

12.3.2 使用实体图

实体图是实体节点和属性的一种声明，以便在执行一个 EntityManager#find()或者在查询操作上使用一个提示时重写或扩充默认抓取计划。下面是使用实体图的检索操作的一个示例：

路径：/examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
Map<String, Object> properties = new HashMap<>();
properties.put(
    "javax.persistence.loadgraph",           ← "Item"
    em.getEntityGraph(Item.class.getSimpleName())
);
Item item = em.find(Item.class, ITEM_ID, properties);
// select * from ITEM where ID = ?
```

正在使用的实体图名称是 Item，并且用于 find()操作的提示表明它应该是加载图。这意味着会将由该实体图的属性节点指定的属性作为 FetchType.EAGER 来处理，并且会根据映射中未指定属性的指定或默认 FetchType 来处理它们。

以下代码就是这个图形的声明以及实体类的默认抓取计划：

路径: /model/src/main/java/org/jpwh/model/fetching/fetchloadgraph/Item.java

```
@NamedEntityGraphs({
    @NamedEntityGraph                ← 默认的“Item”实体图
})
@Entity
public class Item {

    @NotNull
    @ManyToOne(fetch = FetchType.LAZY)
    protected User seller;

    @OneToMany(mappedBy = "item")
    protected Set<Bid> bids = new HashSet<>();

    @ElementCollection
    protected Set<String> images = new HashSet<>();

    // ...
}
```

元数据中的实体图具有名称并且与一个实体类关联；它们通常是在实体类上的注解中声明的。如果愿意的话，可以将它们放在 XML 中。如果不对一个实体图命名，它就会得到其所属实体类的简单名称，此处就是 `Item`。如果不在图形中指定任何属性节点，就像上一个示例中的空实体图那样，则会使用实体类的默认值。在 `Item` 中，所有的关联和集合都是延迟映射的；这就是该默认抓取计划。因此，到目前为止你所做的并没有改变什么，而且没有任何提示的 `find()` 操作将产生相同的结果：`Item` 实例被加载，而 `seller`、`bids` 和 `images` 没有被加载。

或者，可以使用一个 API 来构建一个实体图：

路径: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
EntityGraph<Item> itemGraph = em.createEntityGraph(Item.class);

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", itemGraph);

Item item = em.find(Item.class, ITEM_ID, properties);
```

这同样是一个没有任何属性节点的空实体图，直接指定给一个检索操作。

我们假设你希望编写一个在启用时将 `Item#seller` 的延迟默认改为急抓取的实体图：

路径: /model/src/main/java/org/jpwh/model/fetching/fetchloadgraph/Item.java

```
@NamedEntityGraphs({
    @NamedEntityGraph(
        name = "ItemSeller",
        attributeNodes = {
            @NamedAttributeNode("seller")
        }
    )
})
```


@Entity

```
public class Item {
    // ...
}
```

现在当希望急加载 Item 和 seller 时根据名称启用这个图形：

路径: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
Map<String, Object> properties = new HashMap<>();
properties.put(
    "javax.persistence.loadgraph",
    em.getEntityGraph("ItemSeller")
);

Item item = em.find(Item.class, ITEM_ID, properties);
// select i.*, u.*
// from ITEM i
// inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

如果不希望在注解中硬编码该图形，则可以改用 API 构建它：

路径: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
EntityGraph<Item> itemGraph = em.createEntityGraph(Item.class);
itemGraph.addAttributeNodes(Item_.seller);
Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", itemGraph);
Item item = em.find(Item.class, ITEM_ID, properties);
// select i.*, u.*
// from ITEM i
// inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

静态元模型

到目前为止你只看到了用于 find() 操作的属性。还可为查询启用实体图，作为提示：

路径: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
List<Item> items =
    em.createQuery("select i from Item i")
        .setHint("javax.persistence.loadgraph", itemGraph)
        .getResultList();
// select i.*, u.*
// from ITEM i
// left outer join USERS u on u.ID = i.SELLER_ID
```

实体图可能会很复杂。以下声明显示了如何使用可重用的子图形声明：

路径: /model/src/main/java/org/jpwh/model/fetching/fetchloadgraph/Bid.java

```
@NamedEntityGraphs({
```

```

@NamedEntityGraph(
    name = "BidBidderItemSellerBids",
    attributeNodes = {
        @NamedAttributeNode(value = "bidder"),
        @NamedAttributeNode(
            value = "item",
            subgraph = "ItemSellerBids"
        )
    },
    subgraphs = {
        @NamedSubgraph(
            name = "ItemSellerBids",
            attributeNodes = {
                @NamedAttributeNode("seller"),
                @NamedAttributeNode("bids")
            })
    }
)

@Entity
public class Bid {
    // ...
}

```

在检索 Bid 实例时作为加载图形启用的这个实体图还会触发 Bid#bidder、Bid#item、Item#seller 和所有 Item#bids 的急抓取。尽管可以按照你喜欢的方式随意命名你的实体图，但我们还是建议你制定一个团队中所有人都可以遵循的约定，并将字符串移动到共享的常量中。

使用该实体图 API，之前的计划看起来就会像下面这样：

路径：/examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```

EntityGraph<Bid> bidGraph = em.createEntityGraph(Bid.class);
bidGraph.addAttributeNodes(Bid_.bidder, Bid_.item);
Subgraph<Item> itemGraph = bidGraph.addSubgraph(Bid_.item);
itemGraph.addAttributeNodes(Item_.seller, Item_.bids);

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", bidGraph);

Bid bid = em.find(Bid.class, BID_ID, properties);

```

到目前为止你仅看到了作为加载图的实体图。还有一个选项：可以使用 javax.persistence.fetchgraph 提示将实体图作为抓取图启用。如果用一个抓取图执行 find()或查询操作，那么不在你的计划中的所有属性和集合就会变成 FetchType.LAZY，并且你计划中的所有节点都会是 FetchType.EAGER。这样就有效忽略了实体属性和集合映射中所有的 FetchType 设置，而加载图功能是最唯一的扩充。

JPA 实体图操作的两个弱点需要介绍一下，因为你很快就会碰到它们。首先，你仅可以修改抓取计划，而不能修改 Hibernate 抓取策略(批量/子查询/联结/查询)。其次，在注解

或 XML 中声明一个实体图并非完全是类型安全的：属性名称是字符串。EntityGraph API 至少是类型安全的。

12.4 本章小结

- 抓取配置文件会将抓取计划(应该加载什么数据)和抓取策略(应该如何加载数据)合并在一起，封装在可重用的元数据或代码中。
- 创建了一个全局抓取计划并且定义了应该总是将哪些关联和集合加载到内存中。基于用例、如何访问关联实体与遍历应用程序中的集合，以及哪些数据应该在分离状态中可用来定义抓取计划。
- 介绍了为抓取计划选择合适的抓取策略。目标是 minimized SQL 语句的数量以及每个必须执行的 SQL 语句的复杂性。使用各种优化策略避免 n+1 查询和笛卡尔积问题。
- 探讨了 Hibernate 抓取配置文件和实体图(即 JPA 中的抓取配置文件)。

- 使用 Hibernate Envers 进行审计和版本控制
- 过滤数据

本章将介绍在数据通过 Hibernate 引擎传递时过滤它们的许多不同策略。当 Hibernate 从数据库加载数据时，可以用一个过滤器透明地限制应用程序看到的数据。当 Hibernate 在数据库中存储数据时，可以侦听这样一个事件并且执行一些辅助例程，例如，写入一条审计日志或者为该记录指定一个用户标识符。

我们将研究以下数据过滤功能和 API：

- 在 13.1 节中，将学习响应一个实体实例的状态变更并且将该状态变更与关联的实体级联在一起。例如，在删除一个 User 时，Hibernate 可以传递式地自动地保存所有相关的 BillingDetails。在删除一个 Item 时，Hibernate 可以删除与它关联的所有 Bid 实例。可以使用实体关联和集合映射中的特殊属性来启用此标志 JPA。
- Java 持久化标准包括生命周期回调和事件侦听器。事件侦听器是一个用特殊方法编写的类，在实体实例变更状态时由 Hibernate 调用。比如在 Hibernate 加载它之前或者要从数据库中删除之时。这些回调方法还可位于实体类上并且用特殊注解来标记。这样就可以在迁移发生时为对象设置附加效应。Hibernate 还具有几个专用的扩展点，它们允许在元引擎中从数据库级和拦截生命周期事件。我们将在 13.2 节中探讨这一点。
- 一个常见的附加效应是编写一条审计日志。审计日志通常包含与所修改数据有关的信息，何时做出的修改以及谁做出的修改。一个更复杂的审计系统可能需要存储数据和时态视图的多个版本。你可能希望要求 Hibernate 加载“上局的”数据。这就是我们将在 13.3 节中介绍 Hibernate Envers 时的一个复杂问题。Hibernate Envers 是 JPA 应用程序中用于版本控制和审计的一个子项目。
- 在 13.4 节中，你将看到数据过滤器也可用作专有的 Hibernate API。这些过滤器会请自定义约束添加到 Hibernate 执行的 SQL SELECT 语句中。因此，可以在应用层

数据过滤

13

第 13 章

本章内容简介:

- 级联状态迁移
- 侦听和拦截事件
- 使用 Hibernate Envers 进行审计和版本控制
- 动态过滤数据

本章将介绍在数据通过 Hibernate 引擎传递时过滤它们的许多不同策略。当 Hibernate 从数据库加载数据时，可以用一个过滤器透明地限制应用程序看到的数据。当 Hibernate 在数据库中存储数据时，可以侦听这样一个事件并且执行一些辅助例程：例如，写入一条审计日志或者为该记录指定一个租户标识符。

我们将探究以下数据过滤功能和 API:

- 在 13.1 节中，将学习响应一个实体实例的状态变更并且将该状态变更与关联的实体级联在一起。例如，在保存一个 User 时，Hibernate 可以传递式且自动地保存所有相关的 BillingDetails。在删除一个 Item 时，Hibernate 可以删除与该 Item 关联的所有 Bid 实例。可以使用实体关联和集合映射中的特殊属性来启用此标准 JPA。
- Java 持久化标准包括生命周期回调和事件侦听器。事件侦听器是一个用特殊方法编写的类，在实体实例变更状态时由 Hibernate 调用：比如在 Hibernate 加载它之后或者要从数据库中删除它时。这些回调方法还可位于实体类上并且用特殊注解来标记。这样就可以在迁移发生时执行自定义的附加效应。Hibernate 还具有几个专用的扩展点，它们允许在其引擎中从较低的级别拦截生命周期事件，我们将在 13.2 节中探讨这一点。
- 一个常见的附加效应是编写一条审计日志；这样的日志通常包含与所修改数据有关的信息，何时做出的修改以及谁做出的修改。一个更复杂的审计系统可能需要存储数据和时态视图的几个版本；你可能希望要求 Hibernate 加载“上周的”数据。这就是我们将在 13.3 节中介绍 Hibernate Envers 时的一个复杂问题，Hibernate Envers 是 JPA 应用程序中专用于版本控制和审计的一个子项目。
- 在 13.4 节中，你将看到数据过滤器也可用作专有的 Hibernate API。这些过滤器会将自定义约束添加到 Hibernate 执行的 SQL SELECT 语句中。因此，可以在应用层

中有效定义数据的一个自定义受限视图。例如，可以应用一个通过销售区域或其他任何授权标准来限制所加载数据的过滤器。

我们首先介绍用于传递状态变更的级联选项。

JPA 2 中主要的新功能

- 现在在 JPA 实体事件侦听器类中支持通过 CDI 的依赖注入。

13.1 级联状态迁移

在实体实例变更状态时——比如它由瞬时变成持久化时——关联的实体实例也可以包含在这个状态迁移中。默认不会启用状态迁移的这一级联；每个实体实例都具有独立的生命周期。但对于实体之间的一些关联来说，你可能希望实现细粒度的生命周期依赖。

例如，在 7.3 节中，创建了 Item 和 Bid 实体类之间的一个关联。在这种情况下，不仅在出价被添加到一个 Item 时让它们自动持久化了，而且还会在所属 Item 被删除时自动删除它们。有效地让 Bid 变成了依赖另一个实体 Item 的实体类。

在此关联映射中启用的级联设置是 CascadeType.PERSIST 和 CascadeType.REMOVE。我们还介绍过特殊的开关 orphanRemoval 以及数据库级别的级联删除(具有外键 ON DELETE 选项)如何影响应用程序。

你应该回顾一下这个关联映射及其级联设置；这里我们就不重复介绍了。在这一节中，我们要介绍其他一些很少用到的级联选项。

13.1.1 可用的级联选项

表 13-1 汇总了 Hibernate 中所有可用的级联选项。注意每一个选项是如何与 EntityManager 或 Session 操作链接起来的。

表 13-1 用于实体关联映射的级联选项

选 项	描 述
CascadeType.PERSIST	在使用 EntityManager#persist()存储一个实体实例时，刷新时所有关联的实体实例也都将变得持久化
CascadeType.REMOVE	在使用 EntityManager#remove()删除一个实体实例时，刷新时所有关联的实体实例也都将被移除
CascadeType.DETACH	在使用 EntityManager#detach()从持久化上下文中回收一个实体实例时，所有关联的实体实例也都被分离
CascadeType.MERGE	在使用 EntityManager#merge()将一个瞬时或分离的实体实例合并到持久化上下文中时，所有关联的瞬时或分离的实体实例也都被合并

(续表)

选 项	描 述
CascadeType.REFRESH	在使用 EntityManager#refresh()刷新一个持久化实体实例时，所有关联的持久化实体实例也都将被刷新
org.hibernate.annotations. CascadeType.REPLICATE	在使用 Session#replicate()将一个分离的实体实例复制到数据库中时，所有关联的分离实体实例也都将被复制
CascadeType.ALL	为所映射关联启用所有级联选项的简写

如果有心学习的话，就会找到在 org.hibernate.annotations.CascadeType 枚举中定义的多级联选项。不过，如今唯一相关的选项是 REPLICATE 和 Session#replicate()操作。其他所有 Session 操作都有一个标准等效物或 EntityManager API 上的替代项，因此可以忽略这些设置。

我们已经介绍过 PERSIST 和 REMOVE 选项。我们来看看传递式分离、合并、刷新以及复制。

13.1.2 传递式分离与合并

我们假设你希望从数据库中检索一个 Item 及其 bids，并在分离状态中使用此数据。Bid 类用 @ManyToOne 映射了此关联。它与 Item 中的这个 @OneToMany 集合映射是双向的：

路径：/model/src/main/java/org/jpwh/model/filtering/cascade/Item.java

```
@Entity
public class Item {

    @OneToMany (
        mappedBy = "item",
        cascade = {CascadeType.DETACH, CascadeType.MERGE}
    )
    protected Set<Bid> bids = new HashSet<Bid>();

    // ...
}
```

传递式分离与合并是使用 DETACH 和 MERGE 级联类型来启用的。现在你要加载该 Item 并且初始化其 bids 集合：

路径：/examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(item.getBids().size(), 2);  ← 初始化 bids
em.detach(item);
```

EntityManager#detach()操作被级联了：它从持久化上下文中回收了 Item 实例以及集合中的所有 bids。如果不加载 bids，则不会分离它们(当然，你也可以关闭该持久化上下文，从而有效分离所有加载的实体实例)。

在分离的状态中，要修改 Item#name，创建一个新的 Bid，并且将它与 Item 链接起来：

路径：/examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
item.setName("New Name");

Bid bid = new Bid(new BigDecimal("101.00"), item);
item.getBids().add(bid);
```

由于正在使用分离的实体状态和集合，所以必须格外注意相同性和相等性。正如 10.3 节中所阐释的那样，应该重写 Bid 实体类上的 equals() 和 hashCode() 方法：

路径：/model/src/main/java/org/jpwh/model/filtering/cascade/Bid.java

```
@Entity
public class Bid {

    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (!(other instanceof Bid)) return false;
        Bid that = (Bid) other;

        if (!this.getAmount().equals(that.getAmount()))
            return false;
        if (!this.getItem().getId().equals(that.getItem().getId()))
            return false;
        return true;
    }

    @Override
    public int hashCode() {
        int result = getAmount().hashCode();
        result = 31 * result + getItem().getId().hashCode();
        return result;
    }
    // ...
}
```

当两个 Bid 实例具有相同的数量并且链接到相同 Item 时，它们就是相等的。

在完成分离状态中的修改之后，下一步就是存储这些变更。使用一个新的持久化上下文，合并该分离的 Item 并且让 Hibernate 检测这些变更：

路径：/examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
Item mergedItem = em.merge(item);
// select i.*, b.*
// from ITEM i
// left outer join BID b on i.ID = b.ITEM_ID
// where i.ID = ?

for (Bid b : mergedItem.getBids()) {
```

← ①合并 item

← ②Bid 具有标识符值

```

    assertNotNull(b.getId());
}

em.flush();
// update ITEM set NAME = ? where ID = ?
// insert into BID values (?, ?, ?, ...)

```

← ③检测名称变更

- ① Hibernate 会合并该分离的 item。首先它会检查该持久化上下文是否已经包含一个具有指定标识符值的 Item。在这个例子中，并没有这样的 Item，因此会从数据库中加载该 Item。Hibernate 足够智能，它知道合并期间还需要 bids，因此它会立即在相同 SQL 查询中提取它们。然后 Hibernate 会将该分离的 item 值复制到加载的实例上，并且在持久化状态中返回给你。相同的过程会被应用到每一个 Bid，并且 Hibernate 将检测到其中一个 bids 是新的。
- ② Hibernate 会在合并期间让这个新 Bid 持久化。它现在具有一个指定的标识符值。
- ③ 当刷新该持久化上下文时，Hibernate 会检测到合并期间修改的 Item 的 name。还会存储新的 Bid。

与集合的级联合并是一个强大功能；思考一下，如果没有 Hibernate 实现此功能，你将必须编写许多代码。

在合并时急抓取关联

在前面的示例中我们说过，Hibernate 足够智能，它会在合并一个分离的 Item 时加载 Item#bids 集合。如果为该关联启用 CascadeType.MERGE，那么 Hibernate 就总是会在合并时使用一个 JOIN 急加载实体关联。这在前面的例子中是明智的，其中初始化、分离并且修改了 Item#bids。因此 Hibernate 在用一个 JOIN 进行合并时加载该集合是必须的并且是最合适的。但如果将一个 Item 实例和未初始化的 bids 集合或者未初始化的 seller 代理合并在一起，则 Hibernate 将在合并时使用一个 JOIN 抓取该集合和代理。该合并会初始化它返回的托管 Item 上的这些关联。CascadeType.MERGE 会引起 Hibernate 忽略并有效重写所有 FetchType.LAZY 映射(为 JPA 规范允许)。这个行为在某些情况下可能不是理想的，并且在编写本书时，它并非是可配置的。

我们的下一个示例相对简单，它为相关实体启用了级联刷新。

13.1.3 级联刷新

User 实体类具有一个与 BillingDetails 的一对多关系：该应用程序的每个用户都可以具有几张信用卡、银行账户等。如果不熟悉 BillingDetails 类，可以回顾一下第 6 章中的映射。

可以将 User 和 BillingDetails 之间的关系映射为一个单向一对多实体关联(其中没有 @ManyToOne)：

路径：/model/src/main/java/org/jpwh/model/filtering/cascade/User.java

```

@Entity
@Table(name = "USERS")
public class User {

```

```

@OneToMany(cascade = {CascadeType.PERSIST, CascadeType.REFRESH})
@JoinColumn(name = "USER_ID", nullable = false)
protected Set<BillingDetails> billingDetails = new HashSet<>();

// ...
}

```

为这个关联启用的级联选项是 **PERSIST** 和 **REFRESH**。**PERSIST** 选项简化了结算详情的存储；当将 **BillingDetails** 的一个实例添加到已经持久化的 **User** 集合时，它们会变得持久化。

在 18.3 节中，我们将探讨一个架构，其中持久化上下文可能会开启较长时间，导致上下文中的托管实体实例变得过时。因此，在一些长期运行的会话中，你会希望从数据库中重新加载它们。**REFRESH** 级联选项会确保在重新加载 **User** 实例的状态时，**Hibernate** 也会刷新链接到该 **User** 的每个 **BillingDetails** 实例的状态：

路径：/examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```

User user = em.find(User.class, USER_ID);  ← ①加载 User

assertEquals(user.getBillingDetails().size(), 2);
for (BillingDetails bd : user.getBillingDetails()) {  ← ②初始化集合
    assertEquals(bd.getOwner(), "John Doe");
}

// Someone modifies the billing information in the database!

em.refresh(user);
// select * from CREDITCARD join BILLINGDETAILS where ID = ?
// select * from BANKACCOUNT join BILLINGDETAILS where ID = ?
// select * from USERS
// left outer join BILLINGDETAILS
// left outer join CREDITCARD
// left outer JOIN BANKACCOUNT
// where ID = ?
for (BillingDetails bd : user.getBillingDetails()) {
    assertEquals(bd.getOwner(), "Doe John");
}

```

③刷新 BillingDetails

- ① 从数据库加载一个 **User** 的实例。
- ② 当遍历元素或者调用 `size()` 时，会初始化它的延迟 **billingDetails** 集合。
- ③ 当 `refresh()` 托管的 **User** 实例时，**Hibernate** 会级联对托管 **BillingDetails** 的操作并且用一个 **SQL SELECT** 刷新每一个 **BillingDetails**。如果这些实例不再位于数据库中，那么 **Hibernate** 就会抛出一个 **EntityNotFoundException** 异常。然后，**Hibernate** 会刷新该 **User** 实例并且急加载整个 **billingDetails** 集合以便发现所有新的 **BillingDetails**。

这就是 **Hibernate** 不像它应该的那样智能的情况。首先它会为持久化上下文中以及被集合所引用的每个 **BillingDetails** 实例执行一个 **SQL SELECT**。然后它会再次加载整个集合以便找到所有新增的 **BillingDetails**。显然 **Hibernate** 可以用一个 **SELECT** 完成此任务。

最后一个级联选项用于 **Hibernate** 专用的 `replicate()` 操作。

13.1.4 级联复制

你在 10.2.7 节中第一次看到了复制。Hibernate Session API 上提供了这一非标准操作。主要的用例是将数据从一个数据库复制到另一个。

思考一下 Item 和 User 之间的这个多对一实体关联映射：

路径：/model/src/main/java/org/jpwh/model/filtering/cascade/Item.java

@Entity

```
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "SELLER_ID", nullable = false)
    @org.hibernate.annotations.Cascade(
        org.hibernate.annotations.CascadeType.REPLICATE
    )
    protected User seller;

    // ...
}
```

这里，使用一个 Hibernate 注解启用了该 REPLICATE 级联选项。接下来，要从源数据库中加载一个 Item 及其 seller：

路径：/examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);

assertNotNull(item.getSeller().getUsername()); ← 初始化延迟 Item#seller

tx.commit();
em.close();
```

在关闭该持久化上下文之后，该 Item 和 User 实体实例就处于分离状态了。接下来，连接到数据库并且写入分离的数据：

路径：/examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
tx.begin();
EntityManager otherDatabase = // ... get EntityManager

otherDatabase.unwrap(Session.class)
    .replicate(item, ReplicationMode.OVERWRITE);
// select ID from ITEM where ID = ?
// select ID from USERS where ID = ?

tx.commit();
// update ITEM set NAME = ?, SELLER_ID = ?, ... where ID = ?
// update USERS set USERNAME = ?, ... where ID = ?
otherDatabase.close();
```

当在分离的 Item 上调用 `replicate()` 时, Hibernate 会执行 SQL SELECT 语句来找出 Item 及其 seller 是否已经出现在数据库中。然后,在提交时,当刷新该持久化上下文时, Hibernate 会将 Item 和 seller 的值写入目标数据库中。在前面的示例中,这些行已经存在了,因此你会看到每个行的 UPDATE,重写数据库中的值。如果目标数据库不包含 Item 或 User,这会进行两个 INSERT 操作。

我们要探讨的最后一个级联选项是一个全局设置,以便为所有实体关联启用传递式持久化。

13.1.5 启用全局传递式持久化

无论何时应用程序创建从另一个已经持久化的实例到该实例的引用,如果任何实例变得持久化的话,那么据说一个对象持久化层就会实现可达性持久化。在最纯粹的可达性持久化形式中,数据库具有一些顶层或根对象,可以从中获得所有的持久化对象。理想情况下,如果一个实例不能通过从根持久化对象的引用可达的话,那么它应该变成瞬时的并且从数据库中被删除。

Hibernate 或任何其他 ORM 解决方案都不能实现这一点。实际上,任何 SQL 数据库中都没有类似的根持久化对象,并且没有持久化垃圾回收器可以检测未引用的实例。面向对象的(网络)数据存储可以实现一个垃圾回收算法,类似于 JVM 为内存中对象实现的算法;但这个选项在 ORM 领域不可用,并且为未引用的行对所有表的扫描无法令人满意地执行。

不过,可达性持久化的概念中还有一些值。它有助于你让瞬时实例持久化并且将其状态传递给数据库而无须对持久化管理器进行许多调用。

也可以在 `orm.xml` 映射元数据中为所有的实体关联启用级联持久化,作为持久化单元的默认设置:

路径: `/model/src/main/resources/filtering/DefaultCascadePersist.xml`

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <cascade-persist/>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

Hibernate 现在会将由这个持久化单元映射的域模型中的所有实体关联视作 `CascadeType.PERSIST`。无论何时从一个已经持久化的实体实例创建一个对瞬时实体实例的引用, Hibernate 都会自动让该瞬时实例持久化。

级联选项实际上是对持久化引擎中生命周期事件的预定义反应。如果需要在存储或加载数据时实现一个自定义过程,则可以实现你自己的事件侦听器和拦截器。

13.2 侦听和拦截事件

在这一节中,我们要探讨用于自定义事件侦听器以及 JPA 和 Hibernate 中可用的持久化生命周期拦截器的三种不同 API。可以:

- 使用标准 JPA 生命周期回调方法以及事件侦听器。
- 编写一个专有的 `org.hibernate.Interceptor` 并且在 `Session` 上激活它。
- 用 `org.hibernate.event` SPI 使用 Hibernate 核心引擎的扩展点。

我们先处理标准的 JPA 回调。它们提供了对持久化、加载和移除生命周期事件的轻松访问。

13.2.1 JPA 事件侦听器 and 回调

我们假设你希望在存储新实体实例的任何时候，都发送一个通知邮件给系统管理员。首先，编写一个具有回调方法的生命周期事件侦听器，用 `@PostPersist` 注解，如代码清单 13.1 所示。

代码清单 13.1 在实体实例被存储时通知管理员

路径: `/model/src/main/java/org/jpwh/model/filtering/callback/PersistEntityListener.java`

```
public class PersistEntityListener {    ← ❶ 实体侦听器构造函数

    @PostPersist
    public void notifyAdmin(Object entityInstance) {    ← ❷ 让 notifyAdmin()
                                                        成为一个回调方法

        User currentUser = currentUser.INSTANCE.get();
        Mail mail = Mail.INSTANCE;

        mail.send(
            "Entity instance persisted by "
            + currentUser.getUsername()
            + ": "
            + entityInstance
        );
    }
}
```

❸ 获得用户信息和电子邮件访问

- ❶ 一个实体侦听器类必须不具有构造函数或者公共无参构造函数。它不必实现任何特殊接口。实体侦听器是无状态的；JPA 引擎会自动创建并且销毁它。
- ❷ 可以将任何实体侦听器类的方法注解为用于持久化生命周期事件的回调方法。该 `notifyAdmin()` 方法会在新实体实例被存储到数据库中之后调用。
- ❸ 由于事件侦听器类是无状态的，因此在需要它时可能难以得到更多与上下文有关的信息。此处，你想要当前登录的用户并且访问电子邮件系统来发送一个通知。原始的解决方案是使用线程局部变量和单例；可以在示例代码中找到 `CurrentUser` 和 `Mail` 的源。

实体侦听器类的回调方法具有单个 `Object` 参数：涉及状态变更的实体实例。如果仅为特定实体类型启用回调，则可以声明该参数作为该特定类型。回调方法可能具有任何种类的访问；它不必是公共的。它必须是非静态或最终的，并且不返回任何东西。如果一个回调方法可以抛出未检查过的 `RuntimeException`，则 Hibernate 将终止该操作并且标记当前事

务用于回滚。如果回调方法声明并且抛出一个检查过的 `Exception`，则 `Hibernate` 将包装它并且将它作为 `RuntimeException` 处理。

在事件侦听器类中注入

你通常需要在实现一个事件侦听器时访问与上下文有关的信息和 `API`。前面的示例需要当前登录的用户和一个电子邮件 `API`。基于线程局部和单例的简单解决方案可能并不足以应对较大和更复杂的应用程序。`JPA` 还标准化了与 `CDI` 的集成，因此一个实体侦听器类可以依赖注入和 `@Inject` 注解来访问依赖项。该 `CDI` 容器会在调用侦听器类时提供与上下文有关的信息。注意，即便是使用 `CDI`，你也不能注入当前的 `EntityManager` 来在事件侦听器中访问数据库。我们将在本章稍后探讨用于在一个(`Hibernate`)事件侦听器中访问数据库的不同解决方案。

你仅可以在一个实体侦听器类中使用每个回调注解一次；即，只有一个方法可以被注解为 `@PostPersist`。参见表 13-2 以了解所有可用回调注解的汇总。

表 13-2 生命周期回调注解

注 解	描 述
<code>@PostLoad</code>	在将实体实例加载到持久化上下文中后触发，要么由标识符查找通过导航和代理/集合初始化来加载，要么使用一个查询来加载。也会在刷新一个已经持久化的实例后调用
<code>@PrePersist</code>	在一个实体实例上调用 <code>persist()</code> 时立即调用它。也会在将瞬时状态复制到持久化实例上之后，发现一个实体是瞬时的时候为 <code>merge()</code> 调用它。如果启用 <code>CascadeType.PERSIST</code> ，那么也会为关联的实体调用它
<code>@PostPersist</code>	在用于让实体实例持久化的数据库操作被执行之后调用并且指定一个标识符值。这可能是调用 <code>persist()</code> 或 <code>merge()</code> 的时候，或者因预插入标识符生成器而刷新持久化上下文的时候(参阅 4.2.5 节)。也会启用 <code>CascadeType.PERSIST</code> 时为关联实体调用
<code>@PreUpdate,</code> <code>@PostUpdate</code>	在与数据同步持久化上下文之前和之后执行：即，在刷新之前和之后。仅在实体状态需要同步时触发(例如，由于其被视作是脏的)
<code>@PreRemove,</code> <code>@PostRemove</code>	在调用 <code>remove()</code> 或者实体实例由级联移除时，并且在刷新持久化上下文时删除数据库中记录之后触发

必须为要拦截的任何实体启用一个实体侦听器类，比如这个 `Item`：

路径： `/model/src/main/java/org/jpwh/model/filtering/callback/Item.java`

```
@Entity
@EntityListeners(
    PersistEntityListener.class
)
public class Item {
```

```
// ...
```

如果有几个拦截器的话，`@EntityListeners` 注解可以接受一组侦听器类。如果几个侦听器为相同事件定义回调方法，则 Hibernate 会以声明的顺序调用这些侦听器。另外，可以在 XML 元数据中使用 `<entity>` 的子元素 `<entity-listener>` 来将侦听器类绑定到一个实体。

不必编写一个单独的实体侦听器类来拦截生命周期事件。例如，可以在 `User` 实体类上实现 `notifyAdmin()` 方法：

路径：/model/src/main/java/org/jpwh/model/filtering/callback/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @PostPersist
    public void notifyAdmin(){
        User currentUser = CurrentUser.INSTANCE.get();
        Mail mail = Mail.INSTANCE;
        mail.send(
            "Entity instance persisted by "
                + currentUser.getUsername()
                + ": "
                + this
        );
    }
    // ...
}
```

注意实体类上的回调方法没有任何参数：状态变更中涉及的“当前”实体为 `this`。单个类中不允许存在相同事件的重复回调。但可以在几个侦听器类或一个侦听器和一个实体类中使用回调方法拦截相同事件。

还可以在实体超类上为整个层次结构添加回调方法。如果对于某个特定实体子类，你希望禁用该超类的回调，则可以使用 `@ExcludeSuperclassListener` 注解该子类，或者在 XML 元数据中使用 `<exclude-superclass-listeners>` 映射它。

可以声明默认的实体侦听器类，为持久化单元中的所有实体在 XML 元数据中启用：

路径：/model/src/main/resources/filtering/EventListeners.xml

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener
        class="org.jpwh.model.filtering.callback.PersistEntityListener"/>
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

如果希望为特定实体禁用默认的实体侦听器，则可以在 XML 元数据中使用 `<exclude-`

default-listeners>映射它或者使用@ExcludeDefaultListeners 注解标记它:

路径: /model/src/main/java/org/jpwh/model/filtering/callback/User.java

```
@Entity
@Table(name = "USERS")
@ExcludeDefaultListeners
public class User {

    // ...

}
```

注意启用实体侦听器是附加的。如果启用和/或在 XML 元数据和注解中绑定实体侦听器, 则 Hibernate 将按以下顺序全部调用它们:

- 用于持久化单元的默认侦听器, 按照 XML 元数据中声明的顺序。
- 在一个实体上声明的监听器, 按照指定顺序使用@EntityListeners。
- 首先调用实体超类中声明的回调方法, 从最通用的超类开始。最后是实体类上的回调方法。

JPA 事件侦听器及回调为使用你自己的程序响应生命周期事件提供了一个基础框架。Hibernate 还有一个更细粒度且功能强大的可选 API: org.hibernate.Interceptor。

13.2.2 实现 Hibernate 拦截器

我们假设你希望在一个单独的数据库表中写入数据修改的审计日志。例如, 可以为每一个 Item 记录与创建和更新事件有关的信息。该审计日志包括用户、数据和事件时间、所发生事件的类型以及被修改 Item 的标识符。

通常是使用数据库触发器来处理审计日志。另一方面, 让应用程序来负责这一点有时会更好, 尤其是在需要不同数据库之间的可移植性时。

需要几个元素来实现审计日志。首先, 必须标记希望启用审计日志的实体类。接下来, 你要定义对哪些信息记录日志, 比如用户、日期、时间以及修改类型。最后, 你要使用自动创建审计追踪的 org.hibernate.Interceptor 将其全部结合在一起。

首先, 创建一个标记接口, Auditable:

路径: /model/src/main/java/org/jpwh/model/filtering/interceptor/Auditable.java

```
public interface Auditable {
    public Long getId();
}
```

这个接口要求一个持久化实体类用一个获取方法公开其标识符; 需要这个属性来记录审计追踪。然后为特定持久化类启用审计日志就很简单了。你要将它添加到类的声明, 比如为 Item 所做的那样:

路径: /model/src/main/java/org/jpwh/model/filtering/interceptor/Item.java

```
@Entity
public class Item implements Auditable {
```



```
// ...
```

现在，创建一个新的持久化实体类，`AuditLogRecord`，将你希望记录的信息写入审计数据库表中：

路径：/model/src/main/java/org/jpwh/model/filtering/interceptor/AuditLogRecord.
java

```
@Entity
```

```
public class AuditLogRecord {
```

```
    @Id
```

```
    @GeneratedValue(generator = "ID_GENERATOR")
```

```
    protected Long id;
```

```
    @NotNull
```

```
    protected String message;
```

```
    @NotNull
```

```
    protected Long entityId;
```

```
    @NotNull
```

```
    protected Class entityClass;
```

```
    @NotNull
```

```
    protected Long userId;
```

```
    @NotNull
```

```
    @Temporal(TemporalType.TIMESTAMP)
```

```
    protected Date createdOn = new Date();
```

```
    // ...
```

```
}
```

无论 Hibernate 何时在数据库中插入或更新一个 Item，你都会希望存储 `AuditLogRecord` 的一个实例。Hibernate 拦截器可以自动处理这个任务。相较于在 `org.hibernate.Interceptor` 中实现所有的方法，可以扩展该 `EmptyInterceptor` 并且仅重写所需的方法，如代码清单 13.2 所示。

代码清单 13.2 记录修改事件的 Hibernate 拦截器

路径：/examples/src/test/java/org/jpwh/test/filtering/AuditLogInterceptor.java

```
public class AuditLogInterceptor extends EmptyInterceptor {
```

```
    protected Session currentSession;    ← ① 访问数据库
```

```
    protected Long currentUserId;
```

```
    protected Set<Auditable> inserts = new HashSet<Auditable>();
```

```
    protected Set<Auditable> updates = new HashSet<Auditable>();
```

```
    public void setCurrentSession(Session session) {
```

```
        this.currentSession = session;
```

```
    }
```

```

public void setCurrentUserId(Long currentUserId) {
    this.currentUserId = currentUserId;
}

public boolean onSave(Object entity, Serializable id,
    Object[] state, String[] propertyNames,
    Type[] types)
    throws CallbackException {
    if (entity instanceof Auditable)
        inserts.add((Auditable)entity);
    return false;
}

public boolean onFlushDirty(Object entity, Serializable id,
    Object[] currentState,
    Object[] previousState,
    String[] propertyNames, Type[] types)
    throws CallbackException {
    if (entity instanceof Auditable)
        updates.add((Auditable)entity);
    return false;
}

// ...
}

```

② 当实例被持久化时调用

没有修改该状态

③ 如果实例是脏的，则调用

← 没有修改 currentState

① 需要访问数据库以写入审计日志，因此这个拦截器需要一个 Hibernate Session。你还希望在每条审计日志记录中存储当前登录用户的标识符。inserts 和 updates 实例变量是这个拦截器将在其中持有其内部状态的集合。

② 会在实体实例被持久化时调用这个方法。

③ 会在刷新持久化上下文期间检测到实体实例为脏时调用这个方法。

该拦截器会收集 inserts 和 updates 中修改过的 Auditable 实例。注意在 onSave() 中，可能没有被分配给指定实体实例的标识符值。Hibernate 会确保在刷新期间设置实体标识符，因此会在 postFlush() 回调中写入实际的审计日志追踪，代码清单 13.2 中并没有显示它：

路径：/examples/src/test/java/org/jpwh/test/filtering/AuditLogInterceptor.java

```

public class AuditLogInterceptor extends EmptyInterceptor {
    // ...

    public void postFlush(Iterator iterator) throws CallbackException {
        Session tempSession =
            currentSession.sessionWithOptions()
                .transactionContext()
                .connection()
                .openSession();

        try {

```

① 写入审计日志记录

② 创建临时的 Session

③ 存储 AuditLogRecords

```

    for (Auditable entity : inserts) {
        tempSession.persist(
            new AuditLogRecord("insert", entity, currentUserId)
        );
    }
    for (Auditable entity : updates) {
        tempSession.persist(
            new AuditLogRecord("update", entity, currentUserId)
        );
    }

    tempSession.flush();
} finally {
    tempSession.close();
    inserts.clear();
    updates.clear();
}
}
}

```

← ④ 关闭临时 Session

- ❶ 会在持久化上下文的刷新完成之后调用这个方法。此处，要为之前收集的所有插入和更新写入审计日志记录。
- ❷ 不能访问原始的持久化上下文：当前该 Session 会在这个拦截器中执行。该 Session 在拦截器调用期间处于脆弱状态。Hibernate 会让你使用 `sessionWithOptions()` 方法创建一个新的 Session，它会继承来自原始 Session 的一些信息。这个新的临时 Session 可以像原始 Session 一样使用相同的事务和数据库连接。
- ❸ 要使用该临时 Session 为每一个插入和更新存储一个新的 `AuditLogRecord`。
- ❹ 要独立于原始 Session 刷新和关闭该临时 Session。

现在你已经准备好在创建 `EntityManager` 时用一个 Hibernate 属性启用这个拦截器了：

路径：/examples/src/test/java/org/jpwh/test/filtering/AuditLogging.java

```

EntityManagerFactory emf = JPA.getEntityManagerFactory();

Map<String, String> properties = new HashMap<String, String>();
properties.put(
    org.hibernate.jpa.AvailableSettings.SESSION_INTERCEPTOR,
    AuditLogInterceptor.class.getName()
);

EntityManager em = emf.createEntityManager(properties);

```

启用默认拦截器

如果希望为任何 `EntityManager` 启用默认的拦截器，则可以将 `persistence.xml` 中的 `hibernate.ejb.interceptor` 属性设置到一个实现 `org.hibernate.Interceptor` 的类。注意，不同于限定于会话的拦截器，Hibernate 会共享此默认拦截器，因此它必须是线程安全的！示例 `AuditLogInterceptor` 并非线程安全的。

这个 `EntityManager` 现在具有一个启用的 `AuditLogInterceptor`，但也必须用当前的 `Session` 和登录用户标识符来配置该拦截器。这涉及一些类型转换来访问 `Hibernate API`：

路径：/examples/src/test/java/org/jpwh/test/filtering/AuditLogging.java

```
Session session = em.unwrap(Session.class);
AuditLogInterceptor interceptor =
    (AuditLogInterceptor) ((SessionImplementor)
        session).getInterceptor();
interceptor.setCurrentSession(session);
interceptor.setCurrentUserId(CURRENT_USER_ID);
```

现在就准备好使用 `EntityManager` 了，并且无论何时你用它存储或修改一个 `Item` 实例，都会写入一条审计追踪。

`Hibernate` 拦截器是灵活的，并且，不同于 `JPA` 事件侦听器 and 回调方法，你要在事件发生时访问更多的与上下文有关的信息。话虽如此，但 `Hibernate` 允许你使用它所基于的可扩展事件系统更深入挂接到其核心。

13.2.3 核心事件系统

`Hibernate` 核心引擎是基于事件和侦听器模型的。例如，如果 `Hibernate` 需要保存一个实体实例，它就会触发一个事件。无论侦听此类事件的是谁，都可以捕获该事件并且处理数据的保存。因此 `Hibernate` 会将其所有核心功能作为一组默认侦听器来实现，它们可以处理所有的 `Hibernate` 事件。

在设计上 `Hibernate` 是开放的：可以为 `Hibernate` 事件编写和启用你自己的侦听器。可以替换已有的默认侦听器或者扩展它们并且执行一个附加效应或额外的程序。很少会替换事件侦听器；这样做的话就表明，你自己的侦听器实现会负责一部分 `Hibernate` 核心功能。

实质上，该 `Session` 接口(及其用途更少的同伴 `EntityManager`)的所有方法都与一个事件有关。`find()`和 `load()`方法会触发一个 `LoadEvent`，并且默认情况下这个事件是使用 `DefaultLoadEventListener` 来处理的。

自定义侦听器应该为它希望处理的事件实现合适的接口，并且/或者扩展由 `Hibernate` 提供的其中一个便利基类，或任何默认的事件侦听器。这里有一个自定义加载事件侦听器的示例。见代码清单 13.3。

代码清单13.3 自定义加载事件侦听器

路径：/examples/src/test/java/org/jpwh/test/filtering/SecurityLoadListener.java

```
public class SecurityLoadListener extends DefaultLoadEventListener {

    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
        throws HibernateException {

        boolean authorized =
            MySecurity.isAuthorized(
                event.getEntityClassName(), event.getEntityId()
            );
    }
}
```

```
if (!authorized)
    throw new MySecurityException("Unauthorized access");
super.onLoad(event, loadType);
}
```

这个侦听器会执行自定义的授权代码。实际上侦听器应该被考虑为单例的，这意味着它要在持久化上下文之间共享，并因此不应将任何与事物相关的状态保存为实例变量。要查看原生 Hibernate 中所有在列的事件和侦听器接口，可以阅读 `org.hibernate.event` 包的 API Javadoc。

在 `persistence.xml` 中，要在 `<persistenceunit>` 中为每个核心事件启用侦听器：

路径：/model/src/main/resources/META-INF/persistence.xml

```
<properties>
    <property name="hibernate.ejb.event.load"
        value="org.jpwh.test.filtering.SecurityLoadListener"/>
</properties>
```

该配置设置的属性名称总是以 `hibernate.ejb.event` 开头，后面跟着你希望侦听的事件类型。可以在 `org.hibernate.event.spi.EventType` 中找到所有列出的事件类型。该属性的值可以用逗号分隔的一系列侦听器类名称；Hibernate 会按指定顺序调用每个侦听器。

你很少必须用你自己的功能扩展 Hibernate 核心事件系统。大多数时候，`org.hibernate.Interceptor` 就足够灵活了。它有助于使用更多的选项并且能够以模块化方式替换 Hibernate 核心引擎的任何部分。

你在上一节中看到的审计日志实现非常简单。如果需要记录更多用于审计的信息的话，比如实际修改的实体属性值，则可以考虑使用 Hibernate Envers。

13.3 使用 Hibernate Envers 进行审计和版本控制

Envers 是专用于在数据库中审计日志和保持数据多版本的 Hibernate 套件的一个项目。这类似于你可能已经熟悉的版本控制系统，比如 Subversion 和 Git。

启用 Envers 后，当在应用程序的主表中添加、修改和删除数据时，就会在单独的数据库表中自动存储数据的副本。Envers 会在内部使用你在上一节中看到过的 Hibernate 事件 SPI。Envers 会侦听 Hibernate 事件，并且当 Hibernate 在数据库中存储变更时，Envers 会在其专用的表中创建数据的副本和记录一个版本。

Envers 会将工作单元中(即事务中的所有数据)修改分组为具有一个版本号的修改集。可以使用 Envers API 编写查询以检索指定版本号或时间戳的历史数据：例如，“找出上周五的所有 Item 实例。”首先必须在应用程序中启用 Envers。

13.3.1 启用审计日志

无须做进一步的配置，只要你将其 JAR 文件放到你的类路径上(或者，像本书示例代码中所示的那样，将它包含为一个 Maven 依赖)，就可以使用 Envers 了。要使用 `@org.hibernate.envers.Audited` 注解为一个实体类选择性地启用审计日志。见代码清单 13.4。

代码清单13.4 为Item实体启用审计日志
路径: /model/src/main/java/org/jpwh/model/filtering/envers/Item.java

```
@Entity
@org.hibernate.envers.Audited
public class Item {

    @NotNull
    protected String name;

    @OneToMany(mappedBy = "item")
    @org.hibernate.envers.NotAudited
    protected Set<Bid> bids = new HashSet<Bid>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "SELLER_ID", nullable = false)
    protected User seller;

    // ...
}
```

现在已经为 Item 实例及实体的所有属性启用了审计日志。要为特定属性禁用审计日志，则使用 `@NotAudited` 来注解它。在这个例子中，Envers 会忽略 bids 但审计 seller。你还必须在 User 类上使用 `@Audited` 启用审计。

Hibernate 现在将生成(或期望)额外的数据库表为每个 Item 和 User 持有历史数据。图 13-1 显示了用于这些表的架构。

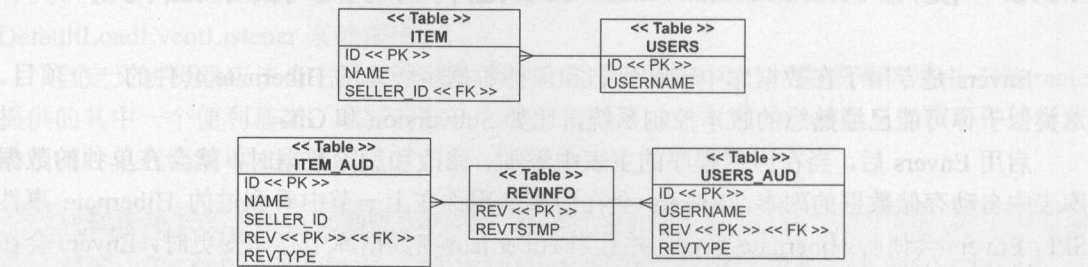


图 13-1 用于 Item 和 User 实体的审计日志表

ITEM_AUD 和 USERS_AUD 表是存储 Item 和 User 实例的修改历史的地方。当修改数据并且提交一个事务时，Hibernate 会将具有时间戳的新版本号插入到 REVINFO 表中。然后，对于变更集中涉及的每个修改过和审计过的实体实例，其数据副本会被存储在审计表中。版本号列上的外键会将变更集链接到一起。REVTYPE 列会持有变更的类型：无论事务中是否插入、更新或删除了实体实例。Envers 不会自动移除任何版本信息或历史数据；

甚至在你删除一个 Item 实例之后，你也仍旧具有其存储在 ITEM_AUD 中的之前版本。

我们运行一些事务来查看这如何运行。

13.3.2 创建审计追踪

在以下代码示例中，你看到了几个涉及 Item 及其 seller(即 User)的事务。你要创建和存储一个 Item 和 User，然后修改它们，随后最终删除该 Item。

你应该已经熟悉这段代码了。Envers 会在使用 EntityManager 时自动创建审计追踪：

路径: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

User user = new User("johndoe");
em.persist(user);

Item item = new Item("Foo", user);
em.persist(item);

tx.commit();
em.close();
```

路径: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
item.setName("Bar");
item.getSeller().setUsername("doejohn");

tx.commit();
em.close();
```

路径: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
em.remove(item);

tx.commit();
em.close();
```

Envers 会通过记录三个变更集来为这一系列事务透明写入审计追踪。要访问该历史数据，首先必须获得该版本号，表示你希望访问的变更集。

13.3.3 找出版本

使用 Envers AuditReader API，就可以找到每个变更集的版本号，见代码清单 13.5。

代码清单13.5 获得变更集的版本号

路径: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```

AuditReader auditReader = AuditReaderFactory.get(em); ← ❶ AuditReader API
Number revisionCreate = ← ❷ 获得版本号
    auditReader.getRevisionNumberForDate(TIMESTAMP_CREATE);
Number revisionUpdate =
    auditReader.getRevisionNumberForDate(TIMESTAMP_UPDATE);
Number revisionDelete =
    auditReader.getRevisionNumberForDate(TIMESTAMP_DELETE);
List<Number> itemRevisions = auditReader.getRevisions(Item.class, ITEM_ID);
assertEquals(itemRevisions.size(), 3);
for (Number itemRevision : itemRevisions) {
    Date itemRevisionTimestamp = auditReader.getRevisionDate(itemRevision);
    // ...
} ← ❸ 找到变更集
    ← ❹ 获得时间戳

List<Number> userRevisions = auditReader.getRevisions(User.class, USER_ID); ← ❺ 统计版本数量
assertEquals(userRevisions.size(), 2);

```

- ❶ 主要的 Envers API 是 AuditReader。可以使用一个 EntityManager 访问它。
- ❷ 给定一个时间戳，可以找到在该时间戳上或之前做出的变更集的版本号。
- ❸ 如果不使用一个时间戳，则可以得到其中涉及特定审计过的实体实例的所有版本号。这一操作会找出所有其中创建、修改或删除了指定 Item 的变更集。在我们的示例中，我们创建、修改、然后删除了该 Item。因此，我们三个版本。
- ❹ 如果有一个版本号，则可以得到 Envers 记录该变更集时的时间戳。
- ❺ 我们创建和修改了该 User，因此有两个版本。

在代码清单 13.5 中，我们假设你要么知道一个事务(大致的)时间戳，要么具有一个实体的标识符值，以便可以获得其版本。如果都不知道的话，那么你可能希望用查询来找到审计日志。如果必须在应用程序的用户界面中显示所有变更集的列表，那么这也是有用的。

以下代码能找出该 Item 实体类的所有版本并且加载每个 Item 版本以及该变更集的审计日志信息：

路径: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```

AuditQuery query = auditReader.createQuery()
    .forRevisionsOfEntity(Item.class, false, false); ← ❶ 查询以获得审计追踪详情

List<Object[]> result = query.getResultList(); ← ❷ 获得审计追踪详情
for (Object[] tuple : result) {
    Item item = (Item) tuple[0];
    DefaultRevisionEntity revision = (DefaultRevisionEntity) tuple[1];
    RevisionType revisionType = (RevisionType) tuple[2];
    ← ❸ 获得版本详情

    if (revision.getId() == 1) { ← ❹ 获得版本类型
        assertEquals(revisionType, RevisionType.ADD);
    }
}

```

```

        assertEquals(item.getName(), "Foo");
    } else if (revision.getId() == 2) {
        assertEquals(revisionType, RevisionType.MOD);
        assertEquals(item.getName(), "Bar");
    } else if (revision.getId() == 3) {
        assertEquals(revisionType, RevisionType.DEL);
        assertNull(item);
    }
}

```

- ❶ 如果不知道修改时间戳或者版本号，则可以使用 `forRevisionsOfEntity()` 编写一个查询以获得特定实体的所有审计追踪详情。
- ❷ 这个查询会将审计追踪详情作为 `Object[]` 的一个 `List` 来返回。
- ❸ 每个结果元组都包含特定版本的实体实例、版本详情(其中包括版本号和修改时间戳)、以及版本类型。
- ❹ 该版本类型表明为何 Envers 创建了该版本，因为在数据库中插入、修改或删除了该实体实例。

版本号是连续递增的；一个较高的版本号总是实体实例最近的版本。现在审计追踪中具有三个变更集的版本号，以便让你可以访问历史数据。

13.3.4 访问历史数据

使用一个版本号，可以访问 `Item` 及其 `seller` 的不同版本。见代码清单 13.6。

代码清单13.6 加载实体实例的历史版本

路径: `/examples/src/test/java/org/jpwh/test/filtering/Envers.java`

❶ 返回审计过的实例

```

Item item = auditReader.find(Item.class, ITEM_ID, revisionCreate);
assertEquals(item.getName(), "Foo");
assertEquals(item.getSeller().getUsername(), "johndoe");

Item modifiedItem = auditReader.find(Item.class, ← ❷ 加载更新过的 Item
    ITEM_ID, revisionUpdate);
assertEquals(modifiedItem.getName(), "Bar");
assertEquals(modifiedItem.getSeller().getUsername(), "doejohn");

Item deletedItem = auditReader.find(Item.class, ← ❸ 处理删除过的 Item
    ITEM_ID, revisionDelete);
assertNull(deletedItem);

User user = auditReader.find(User.class, ← ❹ 返回最接近的版本
    USER_ID, revisionDelete);
assertEquals(user.getUsername(), "doejohn");

```

- ❶ 给定一个版本，`find()` 方法就会返回一个审计过的实体实例版本。此操作会加载 `Item` 被创建之后的版本。

- ❷ 此操作会加载 Item 被更新后的版本。注意这个变更集修改后的 seller 也会如何自动被检索。
- ❸ 在这个版本中，删除了 Item，所以 find()会返回 null。
- ❹ 该示例不会修改这个版本中的 User，因此 Envers 会返回其最接近的历史版本。

AuditReader#find()操作只会检索单个实体实例，就像 EntityManager#find()一样。但所返回的实体实例并非处于持久化状态：该持久化上下文不会管理它们。如果修改 Item 的一个比较老的版本，则 Hibernate 不会更新数据库。考虑分离 AuditReader API 返回的实体实例或者让其只读。

AuditReader 还有一个用于任意查询执行的 API，类似于原生的 Hibernate Criteria API(参阅 16.3 节)。见代码清单 13.7。

代码清单13.7 查询历史的实体实例

路径: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
AuditQuery query = auditReader.createQuery()  
    .forEntitiesAtRevision(Item.class, revisionUpdate);  
  
query.add(  
    AuditEntity.property("name").like("Ba", MatchMode.START)  
);  
  
query.add(  
    AuditEntity.relatedId("seller").eq(USER_ID)  
);  
  
query.addOrder(  
    AuditEntity.property("name").desc()  
);  
  
query.setFirstResult(0);  
query.setMaxResults(10);  
  
assertEquals(query.getResultList().size(), 1);  
Item result = (Item)query.getResultList().get(0);  
assertEquals(result.getSeller().getUsername(), "doejohn");
```

- ❶ 这个查询会返回被限制为特定版本和变更集的 Item 实例。
- ❷ 可以将进一步的限制添加到该查询；此处 Item#name 必须以 “Ba” 开头。
- ❸ 限制可以包括实体关联：例如，正在查找由特定 User 售出的 Item 的版本。
- ❹ 可以对查询结果排序。
- ❺ 可以为查询结果分页。

Envers 支持投影。以下查询只会检索特定版本的 Item#name:

路径: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
AuditQuery query = auditReader.createQuery()  
    .forEntitiesAtRevision(Item.class, revisionUpdate);  
  
query.addProjection(  
    AuditEntity.property("name").like("Ba", MatchMode.START)  
);
```

```

    AuditEntity.property("name")
);

assertEquals(query.getResultList().size(), 1);
String result = (String)query.getSingleResult();
assertEquals(result, "Bar");

```

最终，你可能希望将实体实例回滚到较老的版本。这是通过 `Session#replicate()` 操作和重写已有行来完成的。以下示例会从第一个变更集中加载 `User` 实例，然后用比较老的版本重写数据库中的当前 `User`：

路径：/examples/src/test/java/org/jpwh/test/filtering/Envers.java

```

User user = auditReader.find(User.class, USER_ID, revisionCreate);

em.unwrap(Session.class)
    .replicate(user, ReplicationMode.OVERWRITE);
em.flush();
em.clear();

user = em.find(User.class, USER_ID);
assertEquals(user.getUsername(), "johndoe");

```

`Envers` 还会追踪这个变更作为审计日志中的更新；它只是 `User` 实例的另一个新版本。

时态数据是一个复杂的主题，并且我们支持你为了解更多信息而阅读 `Envers` 参考文档。将详情添加到审计日志，比如做出变更的用户，这并不困难。该文档还显示出，可以如何配置不同的追踪策略并且自定义 `Envers` 使用的数据库架构。

接下来，思考一下你不希望看到数据库中的所有数据的情况。例如，当前登录的应用程序用户可能无权看到全部数据。

通常，你会将一个条件添加到你的查询并且动态限制该结果。这会在你必须处理像安全性这样的问题时变得困难，因为你必须在应用程序中自定义大多数查询。可以用 `Hibernate` 的动态数据过滤器集中和隔离这些限制。

13.4 动态数据过滤器

用于动态数据过滤的第一个用例与数据安全性有关。`CaveatEmptor` 中的一个 `User` 可能具有一个 `ranking` 属性，它是一个简单整数：

路径：/model/src/main/java/org/jpwh/model/filtering/dynamic/User.java

```

@Entity
@Table(name = "USERS")
public class User {

    @NotNull
    protected int rank = 0;

    // ...
}

```

现在假设用户仅可以用相同或较低等级对其他用户提供的商品出价。从业务角度来看,你具有由任意等级(一个数字)定义的几个用户组,并且用户仅可以与具有相同或较低等级的人进行交易。

要实现此需求,必须自定义从数据库加载 `Item` 实例的所有查询。你会检查希望加载的 `Item#seller` 是否具有与当前登录用户对比相同或较低的等级。`Hibernate` 可以使用一个动态过滤器为你完成此任务。

13.4.1 定义动态过滤器

首先,要用过滤器接受的一个名称和动态运行时参数定义过滤器。可以在域模型的任何实体类上或者一个 `package-info.java` 元数据文件中为这个定义放置 `Hibernate` 注解:

路径: `/model/src/main/java/org/jpwh/model/filtering/dynamic/package-info.java`

```
@org.hibernate.annotations.FilterDef(
    name = "limitByUserRank",
    parameters = {
        @org.hibernate.annotations.ParamDef(
            name = "currentUserRank", type = "int"
        )
    }
)
```

这个示例将此过滤器命名为 `limitByUserRank`; 注意该过滤器名称在持久化单元中必须是唯一的。它接受一个类型为 `int` 的运行时参数。如果有几个过滤器定义,则要在 `@org.hibernate.annotations.FilterDefs` 中声明它们。

该过滤器现在是非活动的; 没什么东西表明它应该被应用到 `Item` 实例。必须在希望过滤的类或集合上应用和实现该过滤器。

13.4.2 应用过滤器

在 `Item` 类上应用该定义的过滤器,以便在登录用户没有必需的等级时不会看到商品:

路径: `/model/src/main/java/org/jpwh/model/filtering/dynamic/Item.java`

```
@Entity
@org.hibernate.annotations.Filter(
    name = "limitByUserRank",
    condition =
        ":currentUserRank >= (" +
            "select u.RANK from USERS u " +
            "where u.ID = SELLER_ID" +
        ")"
)
public class Item {
    // ...
}
```


这个 condition 是直接传递到数据库系统的一个 SQL 表达式，以便可以使用任何 SQL 操作符或函数。如果一条记录应该通过该过滤器，则它必须评估为 true。在这个示例中，你要使用一个子查询来获得该商品卖家的 rank。不合格的列，比如 SELLER_ID，涉及被映射到实体类的表。如果当前登录的用户等级不大于或等于由该子查询返回的等级，那么该 Item 实例就会被过滤掉。可以通过在 @org.hibernate.annotations.Filters 中分组它们来应用几个过滤器。

如果为特定工作单元启用了定义过且应用过的过滤器，这会过滤掉未通过条件判定的所有 Item 实例。我们来启用它。

13.4.3 启用过滤器

你已经定义了一个数据过滤器并且将它应用到一个持久化实体类。它仍旧不会过滤任何东西——必须使用 Session API 在应用程序中为特定工作单元启用并且参数化它：

路径: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
org.hibernate.Filter filter = em.unwrap(Session.class)
    .enableFilter("limitByUserRank");

filter.setParameter("currentUserRank", 0);
```

你要通过名称来启用该过滤器；该方法会返回一个你动态设置了运行时参数的 Filter。你必须设置已经定义了的参数；此处它被设置为等级 0。然后这个示例会用这个 Session 中的一个较高等级过滤掉 User 售出的 Items。

该 Filter 的其他有用方法是 getFilterDefinition() (它允许你遍历参数名称和类型) 和 validate() (如果忘记设置一个参数，则它会抛出一个 HibernateException)。还可以用 setParameterList() 设置一系列参数；如果 SQL 限制包含一个具有量词操作符的表达式，那么这就是非常有用的(比如 IN 操作符)。

现在，你在所过滤的持久化上下文上执行的每一个 JPQL 或标准查询都会限制所返回的 Item 实例：

路径: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM where 0 >=
// (select u.RANK from USERS u where u.ID = SELLER_ID)
```

路径: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();
criteria.select(criteria.from(Item.class));
List<Item> items = em.createQuery(criteria).getResultList();
// select * from ITEM where 0 >=
// (select u.RANK from USERS u where u.ID = SELLER_ID)
```

注意 Hibernate 如何动态地将该 SQL 限制条件附加到所生成的语句。

当首次尝试使用动态过滤器时，你很可能会碰到使用标识符检索的问题。你可能期望 `em.find(Item.class, ITEM_ID)` 也会被过滤。不过，情况并非如此：Hibernate 不会将过滤器应用到通过标识符操作进行的检索。其中一个原因是，数据过滤条件都是 SQL 片段，并且通过标识符查找可能是完全在内存中的一级持久化上下文缓存里完成的。类似的论证也适用于多对一和一对一关联的过滤。如果过滤了多对一关联(例如，通过调用 `anItem.getSeller()` 返回 `null`)，则该关联的多样性将会变更！你不会知道该商品是否具有一个卖家或你是否不被允许看到它。

但可以动态过滤集合访问。记住，持久化集合是一个查询的简写法。

13.4.4 过滤集合访问

直到现在，调用 `someCategory.getItems()` 已经返回了被该 `Category` 引用的所有 `Item` 实例。这可以被应用到一个集合的过滤器所限制：

路径：/model/src/main/java/org/jpwh/model/filtering/dynamic/Category.java

@Entity

```
public class Category {
```

```
    @OneToMany(mappedBy = "category")
```

```
    @org.hibernate.annotations.Filter(  
        name = "limitByUserRank",  
        condition =  
            ":currentUserRank>= (" +  
                "select u.RANK from USERS u " +  
                "where u.ID = SELLER_ID" +  
            "),"
```

```
    )  
    protected Set<Item> items = new HashSet<Item>();  
    // ...  
}
```

如果现在在一个 `Session` 中启用该过滤器，则会过滤所有对 `Category#items` 集合的遍历：

路径：/examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
filter.setParameter("currentUserRank", 0);
```

```
Category category = em.find(Category.class, CATEGORY_ID);
```

```
assertEquals(category.getItems().size(), 1);
```

如果当前用户的等级为 0，那么当访问该集合时就只会加载一个 `Item`。现在，使用 100 这个等级，你会看到更多的数据：

路径：/examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
filter.setParameter("currentUserRank", 100);
```

```
category = em.find(Category.class, CATEGORY_ID);
```

```
assertEquals(category.getItems().size(), 2);
```

你可能注意到了，用于两个过滤器应用程序的 SQL 条件是相同的。如果该 SQL 限制对于所有过滤器应用程序是相同的，那么可以在定义该过滤器时设置它作为默认条件，这样就不必重复它了：

路径：/model/src/main/java/org/jpwh/model/filtering/dynamic/package-info.java

```
@org.hibernate.annotations.FilterDef(  
    name = "limitByUserRankDefault",  
    defaultCondition=  
        ":currentUserRank>= (" +  
            "select u.RANK from USERS u " +  
            "where u.ID = SELLER_ID" +  
            ")",  
    parameters = {  
        @org.hibernate.annotations.ParamDef(  
            name = "currentUserRank", type = "int"  
        )  
    }  
)
```

对于动态数据过滤器来说有其他许多极好的用例。你已经看到了指定任意与安全相关的条件的数据访问限制。这可以是用户等级、用户必须归属的特定分组，或者用户已经被分配的角色。可能会用一个区域编码来存储数据(例如，一个销售团队的所有业务联系人)。甚或每个销售人员都只处理涵盖他们区域的数据。

13.5 本章小结

- 级联状态迁移是对持久化引擎中生命周期事件的预定义响应。
- 介绍了与侦听和拦截事件有关的内容。实现了事件侦听器 and 拦截器以便在 Hibernate 加载和存储数据时添加自定义逻辑。还介绍了 JPA 的事件侦听器回滚和 Hibernate 的拦截器扩展点，以及 Hibernate 核心事件系统。
- 可以将 Hibernate Envers 用于审计日志并且在数据库中保持数据的多个版本(就像版本控制系统一样)。使用 Envers，数据的一个副本就会在添加、修改或删除应用程序表中的数据时被自动存储在单独的数据库表中。Envers 会在一个事务中使用版本号将所有的数据修改分组为变更集。然后可以查询 Envers 以检索历史数据。
- 使用动态数据过滤器，Hibernate 就可以自动将任意 SQL 限制附加到它生成的查询。

第 IV 部分

编写查询

本章将介绍：
• 查询 API

本部分将详尽介绍数据查询功能并且涵盖查询语言和 API。这个部分中的所有章节并非都是以教程形式编写的；我们期望你在构建一个应用程序并且为特定查询问题查找一个解决方案时经常阅读本书的这一部分。

从第 14 章开始，我们将使用基本查询 API 探讨创建和执行查询，以便准备和执行查询，以及优化查询的执行。继而在第 15 章中，我们将介绍查询语言，以便编写 JPQL 和条件查询、使用联结有效检索数据并且报告查询和子查询。第 16 章会更深入地介绍高级查询选项：转换查询结果、过滤集合，以及使用 Hibernate API 按条件查询。最后，第 17 章会介绍自定义技术，比如回退到 JDBC、映射 SQL 查询结果、自定义 CRUD 操作，以及调用存储过程。

在阅读完本部分之后，你就能够使用各种查询技术得到你想从数据库中取出的任何数据，以便按需自定义访问。

在本章中，我们将介绍如何使用 JPA 与 Hibernate API 创建和执行查询。这些查询会尽可能简单，以便可以专注于创建和执行 API，而不会受到不熟悉语言的困扰。下一章将介绍查询语言。

我们使用所有 API，在执行之前，必须在应用程序代码中准备好一个查询。有三个不同的步骤：

- (1) 使用任意你希望检索的数据的选择、限制和投影来创建该查询。
- (2) 准备好该查询，将运行时参数绑定到查询参数、设置提示以及设置其他选项，可以随变量的设置重用该查询。
- (3) 对数据库执行准备好的查询并且检索数据，可以控制查询如何被执行以及数据如何被检索到内存中(比如一次性全部检索或一部分一部分地检索)。

创建和执行查询

第 14 章

14

本章内容简介：

- 基础查询 API
- 创建和准备查询
- 优化查询执行

如果使用手动编写的 SQL 已经数年了，则你可能会关注，ORM 将带走你已经习惯的一些表达性和灵活性。使用 Hibernate 和 Java 持久化并没有这种情况。

使用 Hibernate 和 Java 持久化的强有力查询工具，就可以表达通常需要在 SQL 中表达的几乎一切内容，不过按照面向对象的术语来说——就是使用类和类的属性。再者，总是可以回退到 SQL 字符串并且让 Hibernate 完成处理查询结果的繁重任务。对于额外的 SQL 资源，可以查阅我们的参考文献一节。

JPA 2 中主要的新特性

- 现在提供了用于查询编程式创建的类型安全的标准 API。
- 现在可以使用新的 TypedQuery 接口预先声明一个查询结果的类型。
- 可以程式化地保存一个 Query(JPQL、条件或原生 SQL)，以便稍后用作一个指定查询。
- 除了能够设置查询参数、提示、最大结果以及刷新和锁模式之外，JPA 2 使用各种获取方法扩展了 Query API，以便获得当前设置。
- JPA 现在标准化了几个查询提示(超时、缓存使用)。

在本章中，我们将介绍如何使用 JPA 与 Hibernate API 创建和执行查询。这些查询会尽可能简单，以便可以专注于创建和执行 API，而不会受到不熟悉语言的潜在干扰。下一章将介绍查询语言。

共用于所有 API，在执行之前，必须在应用程序代码中准备好一个查询。有三个不同的步骤：

- (1) 使用任意你希望检索的数据的选择、限制和投影来创建该查询。
- (2) 准备好该查询：将运行时参数绑定到查询参数、设置提示以及设置分页选项。可以用变更的设置重用该查询。
- (3) 对数据库执行该准备好的查询并且检索数据。可以控制查询如何被执行以及数据应该如何被检索到内存中(比如一次性全部检索或一部分一部分地检索)。

取决于使用的查询选项，用于查询创建的起始点要么是 `EntityManager`，要么是原生 `Session API`。首先是创建该查询。

14.1 创建查询

JPA 使用 `javax.persistence.Query` 或 `javax.persistence.TypedQuery` 实例来表示一个查询。要使用 `EntityManager#createQuery()` 方法及其变体创建查询。可以在 Java 持久化查询语言 (JPQL) 中编写查询，使用 `CriteriaBuilder` 和 `CriteriaQuery` API 构造它，或者使用普通的 SQL (还有 17.4 节中介绍的 `javax.persistence.StoredProcedureQuery` 可用)。

Hibernate 有其自己的、比较旧的 API 来表示查询：`org.hibernate.Query` 和 `org.hibernate.SQLQuery`。我们稍后将讨论与这些相关的更多内容。首先介绍 JPA 标准接口和查询语言。

14.1.1 JPA 查询接口

假设你希望从数据库中检索所有的 `Item` 实体实例。使用 JPQL，这个简单查询字符串看起来就会有一点点像你熟悉的 SQL：

```
Query query = em.createQuery("select i from Item i");
```

JPA 提供程序会返回一个刷新的 `Query`；到目前为止，Hibernate 还未发送任何 SQL 到数据库。记住，进一步的准备工作以及查询的执行是在单独的步骤中完成的。

JPQL 是紧凑的并且对于具有 SQL 经验的任何人都是熟悉的。相较于表和列名称，JPQL 要依赖实体类和属性名称。除了这些类和属性名称之外，JPQL 是大小写不敏感的，因此你是编写 `SeLEct` 还是 `select` 并没有什么关系。

JPQL (和 SQL) 查询字符串在你的代码中可以是简单的 Java 文字，正如你在上一个示例中所看到的。另外，尤其是在较大的应用程序中，可以从数据访问代码中将查询字符串取出并且移动到注解或 XML 中。然后使用 `EntityManager#createNamedQuery()` 通过名称来访问一个查询。我们将在本章稍后的内容中单独探讨外部化查询；有许多选项可以考虑。

JPQL 的一个重大缺点会作为域模型重构期间的问题暴露出来：如果重命名该 `Item` 类，那么你的 JPQL 查询将被破坏 (不过，有些 IDE 可以检测并重构 JPQL 字符串)。

JPA 和查询语言：HQL 对比 JPQL

在 JPA 出现之前 (即便是在如今的一些文档中)，Hibernate 的查询语言就被称为 HQL。现在 JPQL 和 HQL 之间的区别并不明显。无论你何时使用 `EntityManager` 或 `Session` 将一个查询字符串提供给 Hibernate 中的任何查询接口，它都是一个 JPQL/HQL 字符串。相同的引擎会在内部解析该查询。基础语法和语义是相同的，尽管 Hibernate 像往常一样支持一些特殊的未在 JPA 中标准化过的构造。我们将在一个示例中的特定关键字或子句仅适用于 Hibernate 时告知你。为了简化你的工作，无论何时你看到 HQL 都可以考虑一下 JPQL。

可以使用 `CriteriaBuilder` 和 `CriteriaQuery` API 让查询构造完全类型安全。JPA 也会调用这个条件查询：


```
CriteriaBuilder cb = em.getCriteriaBuilder();  
// Also available on EntityManagerFactory:  
// CriteriaBuilder cb = entityManagerFactory.getCriteriaBuilder();  
  
CriteriaQuery criteria = cb.createQuery();  
criteria.select(criteria.from(Item.class));  
  
Query query = em.createQuery(criteria);
```

首先通过调用 `getCriteriaBuilder()` 来从 `EntityManager` 中得到一个 `CriteriaBuilder`。如果还没准备好一个 `EntityManager`，可能是因为你希望从特定持久化上下文中单独创建该查询，那么可以从通常全局共享的 `EntityManagerFactory` 中获得 `CriteriaBuilder`。

然后使用该构造器创建任意数量的 `CriteriaQuery` 实例。每个 `CriteriaQuery` 至少都有一个用 `from()` 指定的根类；在最后一个示例中，它就是 `Item.class`。这被称为选择；我们将在下一章中更多地探讨它。所示的查询会从数据库中返回所有的 `Item` 实例。

`CriteriaQuery` API 在应用程序中看起来会是无缝的，没有字符串操作。这是你无法在开发时完全指定该查询并且应用程序必须在运行时动态创建它时的最佳选择。想象一下，必须在应用程序中使用许多复选框、输入框和用户可以启用的开关来实现一个搜索蒙板界面。必须从用户选中的搜索选项中动态创建一个数据库查询。使用 JPQL 和字符串连接，会难以编写和维护这样的代码。

可以使用静态的 JPA 元模型编写强类型的 `CriteriaQuery` 调用，不带有字符串。这意味着你的查询将是安全的并且包含在重构操作中。

创建一个分离的条件查询

你总是需要一个 `EntityManager` 或 `EntityManagerFactory` 来得到 `JPACriteriaBuilder`。使用较老的原生 `org.hibernate.Criteria` API，你就只需要访问根实体类来创建一个分离查询，如 16.3 节中所示。

如果需要使用特定于数据库产品的功能，那么你的唯一选择就是原生 SQL。可以在 JPA 中直接执行 SQL 并且使用 `EntityManager#createNativeQuery()` 方法让 Hibernate 处理结果：

```
Query query = em.createNativeQuery(  
    "select * from ITEM", Item.class  
);
```

在执行此 SQL 查询之后，Hibernate 会读取 `java.sql.ResultSet` 并且创建托管 `Item` 实体实例的一个 List。当然，构造一个 `Item` 的所有必要列在结果中都必须是可用的，并且如果 SQL 查询未正确返回它们，则会抛出一个错误。

实际上，应用程序中的大部分查询都会是微不足道的——在 JPQL 中或使用 `CriteriaQuery` 能轻易表达。然后，可能在优化期间，你会找到少量复杂和对性能要求很高的查询。你可能必须使用特殊且专有的 SQL 关键字来控制 DBMS 产品的优化器。然后大多数开发人员会编写 SQL 替代 JPQL 并且将这样的复杂查询移动到一个 XML 文件中，其中，借助 DBA 的帮助，可以从 Java 代码中独立修改它们。Hibernate 仍旧可以为你处理该查询结果；因此你要将 SQL 集成到你的 JPA 应用程序中。在 Hibernate 中使用 SQL 没什么

错；不要让某种形式的 ORM “纯粹性”挡你的路。当有一个特殊用例时，不要试图隐藏它，而是适当对其公开和文档化，以便后继工程师能理解正在发生什么。

在某些情况下，指定从查询中返回的数据类型是有用的。

14.1.2 类型化查询结果

我们假设给定单个 Item 的标识符值，希望使用一个查询只检索该 Item：

```
Query query = em.createQuery(
    "select i from Item i where i.id = :id"
).setParameter("id", ITEM_ID);

Item result = (Item) query.getSingleResult();
```

在这个示例中，你会看到参数绑定和查询执行的一个预览。重要的一点是 `getSingleResult()` 方法的返回值。它是 `java.lang.Object`，并且必须将它转换成一个 `Item`。

如果在创建该查询时提供返回值的类，就可以跳过该转换。这是 `javax.persistence.TypedQuery` 接口的任务：

```
TypedQuery<Item> query = em.createQuery(
    "select i from Item i where i.id = :id", Item.class
).setParameter("id", ITEM_ID);

Item result = query.getSingleResult();  ← 无须强制转换
```

条件查询也支持 `TypedQuery` 接口：

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(cb.equal(i.get("id"), ITEM_ID));

TypedQuery<Item> query = em.createQuery(criteria);

Item result = query.getSingleResult();  ← 无须强制转换
```

注意此 `CriteriaQuery` 并非完全类型安全：`Item#id` 属性是由 `get("id")` 中的一个字符串来处理。在第 3 章中，你看到了如何才能用静态元模型类让这样的查询完全类型安全。

Hibernate 甚至比 JPA 的第一个版本还要老，因此它也有其自己的查询 API。

Hibernate 特性

14.1.3 Hibernate 的查询接口

Hibernate 自己的查询表示形式就是 `org.hibernate.Query` 和 `org.hibernate.SQLQuery`。像往常一样，它们提供的会比 JPA 中标准化的更多，但会损害可移植性。它们也比 JPA 旧得多，因此存在一些功能重叠。

学习 Hibernate 的查询 API 的起始点是 `Session`：

```
Session session = em.unwrap(Session.class);
org.hibernate.Query query = session.createQuery("select i from Item i");
// Proprietary API: query.setResultTransformer(...);
```

你要在标准 JPQL 中编写查询字符串。对比 `javax.persistence.Query`, `org.hibernate.Query` API 具有一些额外的专有方法, 它们仅在 **Hibernate** 中可用。你将在本章稍后以及后续几章中看到与该 API 有关的更多内容。

Hibernate 也有其自己的使用 `org.hibernate.SQLQuery` 的 SQL 结果映射实用工具:

```
Session session = em.unwrap(Session.class);
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select {i.*} from ITEM {i}"
).addEntity("i", Item.class);
```

这个示例要借助 SQL 字符串中的占位符来将 `java.sql.ResultSet` 的列映射到实体属性。我们将在 17.2 节中探讨更多与使用此专有且标准 JPA 结果映射的 SQL 查询集成有关的内容。

Hibernate 还有一个比较老的、专有的 `org.hibernate.Criteria` 查询 API:

```
Session session = em.unwrap(Session.class);
org.hibernate.Criteria query = session.createCriteria(Item.class);
query.add(org.hibernate.criterion.Restrictions.eq("id", ITEM_ID));

Item result = (Item) query.uniqueResult();
```

给定一个 `javax.persistence.Query`, 还可以首先通过解包一个 `org.hibernate.jpa.HibernateQuery` 来访问专有的 **Hibernate** 查询 API:

```
javax.persistence.Query query = em.createQuery(
    // ...
);
org.hibernate.Query hibernateQuery =
    query.unwrap(org.hibernate.jpa.HibernateQuery.class)
        .getHibernateQuery();
hibernateQuery.getQueryString();
hibernateQuery.getReturnAliases();
// ... other proprietary API calls
```

我们专注于标准的 API, 并且稍后介绍一些很少需要的仅可在 **Hibernate** 的 API 中可用的高级选项, 比如滚动鼠标指针和通过示例查询。

在编写你的查询之后, 以及在执行它之前, 你通常会希望通过设置适用于特定执行的参数来进一步准备好该查询。

14.2 准备查询

一个查询具有几个方面：它会定义应该从数据库加载哪些数据以及应用哪些限制，比如 Item 的标识符或者 User 的名称。当编写一个查询时，不应使用字符串连接将这些参数编码到查询字符串中。相反应该使用参数占位符，然后在执行前绑定参数值。这使得可以在保持免受 SQL 注入攻击的同时重新使用具有不同参数值的查询。

根据用户接口，还会需要频繁地分页。要限制通过查询从数据库返回的行数。例如，可能希望仅返回结果 1 到 20 行，因为只能在每一屏上显示这么多数据，然后稍晚一些你会想要返回 21 到 40 行，以此类推。

我们现在开始处理参数绑定。

14.2.1 防止 SQL 注入攻击

如果没有运行时参数绑定，就不得不编写糟糕的代码：

```
String searchString = getValueEnteredByUser(); ←—— 绝不要这样做

Query query = em.createQuery(
    "select i from Item i where i.name = '" + searchString + "'"
);
```

不应该编写像这样的代码，因为恶意用户可以有目的地编写一个搜索字符串在你未期望或不想要的数据库上执行代码——即，通过在搜索对话框中将 searchString 的值输入为 foo' and callSomeStoredProcedure() and 'bar' = 'bar。

正如你所见，原始的 searchString 不再是一个简单的字符串搜索，还会执行数据库中的一个存储过程！引号字符未被转义；因此对存储过程的调用是该查询中的另一个有效表达。如果编写一个像这样的查询，那么就会由于允许在数据库上执行任意代码而让应用程序具有一个主要安全漏洞。这就是一个 SQL 注入攻击。不要将用户输入的未经检查的值传到数据库！幸运的是，一个简单的机制就能避免此错误。

JDBC API 包括用于将值安全绑定到 SQL 语句的功能。它清楚地知道要转义的参数值中的哪些字符，因此前面所说的漏洞就不存在了。例如，数据库驱动会转义给定 searchString 中的单引号字符，并且不再将它们作为控制字符而是作为搜索字符串值的一部分来处理。此外，当使用参数时，数据库就可以有效缓存预编译的准备好的语句，以显著提高性能。

绑定参数有两种方法：命名参数和定位参数。JPA 支持这两种选项，但你无法同时为某个查询使用这两者。

14.2.2 绑定命名参数

使用命名参数，可以将上一节中的查询做如下重写：

```
String searchString = // ...

Query query = em.createQuery(
```

```
"select i from Item i where i.name = :itemName"
).setParameter("itemName", searchString);
```

后面跟着一个参数名称的冒号表示一个命名参数，这里是 `itemName`。在第二步中，要将一个值绑定到 `itemName` 参数。该代码更加整洁、更加安全，并且能更好地执行，因为如果只有参数值发生变化，就可以重用单个编译过的 SQL 语句。

可以从一个 `Query` 中得到 `Parameters` 的一个 `Set`，以便获得关于每个参数(比如名称或必要的 Java 类型)的更多信息，或者验证你是否在执行前正确绑定了所有的参数：

```
for (Parameter<?> parameter : query.getParameters()) {
    assertTrue(query.isBound(parameter));
}
```

`setParameter()` 方法是一个能绑定所有类型参数的通用操作。它只需要借助一点时态类型的帮助：

```
Date tomorrowDate = // ...

Query query = em.createQuery(
    "select i from Item i where i.auctionEnd > :endDate"
).setParameter("endDate", tomorrowDate, TemporalType.TIMESTAMP);
```

Hibernate 需要知道你是否只想要绑定的日期或时间或完全的时间戳。

为方便起见，也可以将一个实体实例传递给 `setParameter()` 方法：

```
Item someItem = // ...

Query query = em.createQuery(
    "select b from Bid b where b.item = :item"
).setParameter("item", someItem);
```

Hibernate 会绑定指定 `Item` 的标识符值。稍后你将看到 `b.item` 被用作 `b.item.id` 的简写。对于条件查询，有一长一短两种方式来绑定参数：

```
String searchString = // ...

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);

Query query = em.createQuery(
    criteria.select(i).where(
        cb.equal(
            i.get("name"),
            cb.parameter(String.class, "itemName")
        )
    )
).setParameter("itemName", searchString);
```

这里将 `String` 类型的 `itemName` 参数占位符放入了 `CriteriaQuery` 中，然后像往常一样

使用 `Query#setParameter()` 方法将一个值绑定到它。

另外，使用 `ParameterExpression`，就不必命名该占位符，并且参数的绑定是类型安全的(不能将一个 `Integer` 绑定到 `ParameterExpression<String>`):

```
String searchString = // ...

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery(Item.class);
Root<Item> i = criteria.from(Item.class);

ParameterExpression<String> itemNameParameter =
    cb.parameter(String.class);

Query query = em.createQuery(
    criteria.select(i).where(
        cb.equal(
            i.get("name"),
            itemNameParameter
        )
    )
).setParameter(itemNameParameter, searchString);
```

用于值绑定的一个很少使用且不太安全的选项是定位查询参数。

14.2.3 使用定位参数

如果愿意，可以使用定位参数来替代命名参数：

```
Query query = em.createQuery(
    "select i from Item i where i.name like ?1 and i.auctionEnd > ?2"
);
query.setParameter(1, searchString);
query.setParameter(2, tomorrowDate, TemporalType.TIMESTAMP);
```

在这个示例中，定位参数标记就是有索引的?1 和?2。你可能从 JDBC 中知道了此类参数占位符，但不具有数字而仅仅有问号标记。JPA 要求你从 1 开始枚举占位符。

提示：在 Hibernate 中，这两种方式都可行，因此要当心！如果使用只有问号标记的 JDBC 样式的定位参数，则 Hibernate 将发出易损坏查询的警告。

我们的建议是避免使用定位参数。如果程式构建复杂查询，它们可能会更加方便，但对于该目的来说，`CriteriaQuery` API 是更好的替代项。

在将参数绑定到你的查询之后，你可能希望在不能一次性显示所有结果时启用分页。

14.2.4 对大结果集分页

处理大结果集通常使用的技术是分页。用户可以看到作为一个页面的搜索请求的结果(例如，对于特定商品)。此页面会每次显示一个受限的子集(比如，10 个商品)，并且用户

可以手动导航到下一页和上一页以便浏览其余的结果。

Query 接口支持查询结果的分页。在这个查询中，请求的页面是从结果集的中间开始的：

```
Query query = em.createQuery("select i from Item i");
query.setFirstResult(40).setMaxResults(10);
```

从第 40 行开始，你检索了后面的 10 行。对 `setFirstResult(40)` 的调用会从第 40 行开始显示结果集。对 `setMaxResults(10)` 的调用会将查询结果集限制为由数据库返回的 10 行。由于 SQL 中没有表示分页的标准方式，因此 Hibernate 知道让分页在特定 DBMS 上有效运行的技巧。

分页在 SQL 级别的结果行上操作是至关重要的。将一个结果限制为 10 行不一定与将结果限制为 Item 的 10 个实例相同！在 15.4.5 节中，你将看到一些使用动态抓取的查询，这些动态抓取无法与 SQL 级别基于行的分页结合使用。我们将再次探讨这个问题。

甚至可以将此灵活的分页选项添加到一个 SQL 查询：

```
Query query = em.createNativeQuery("select * from ITEM");
query.setFirstResult(40).setMaxResults(10);
```

Hibernate 将重写 SQL 查询以包含必要的关键字和子句，以便用于将返回行的数量限制到你指定的页面。

实际上，你会频繁地将一个特殊的计数查询与分页结合使用。如果显示一页商品，则还要让用户知道商品的总数。此外，需要此信息来判定是否有更多的页面要显示以及用户是否可以单击到下一页。这通常需要两个稍微不同的查询：例如，结合使用 `select i from Item i` 和 `setMaxResults()` 以及 `setFirstResult()` 会检索一页商品，并且 `select count(i) from Item i` 会检索可用商品的总数。

Hibernate 特性

为何两个几乎相同的查询是你应该避免的开销。一个流行的技巧是只编写一个查询，但首先使用一个数据库游标执行它以便得到结果的总计数：

```
Query query = em.createQuery("select i from Item i");
org.hibernate.Query hibernateQuery =
    query.unwrap(org.hibernate.jpa.HibernateQuery.class).getHibernateQuery();
org.hibernate.ScrollableResults cursor =
    hibernateQuery.scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);
cursor.last();
int count = cursor.getRowNumber() + 1;
cursor.close();
query.setFirstResult(40).setMaxResults(10);
```

❶ 开箱 Hibernate API 以便使用可滚动游标。

❷ 执行具有数据库游标的查询；这不会将结果集检索到内存中。

- ❸ 跳到数据库中的最后一行结果，然后得到行数。因为行数是从零开始的，所以加 1 以便得到行的总计数。
- ❹ 必须关闭数据库游标。
- ❺ 再次执行该查询，并且检索任何一页数据。

使用此便利策略有一个严重问题：JDBC 驱动和/或 DBMS 可能不支持数据库游标。甚至更糟糕的是，游标看起来可行，但数据会被静默地检索到应用程序内存中；游标并非在数据库上直接操作。大家都知道，Oracle 和 MySQL 驱动会产生问题，我们将在下一节中更多地探讨滚动和游标。稍后在本书的 19.2 节中，我们将进一步探讨一个应用程序环境中的分页策略。

现在查询就准备好执行了。

14.3 执行查询

一旦已经创建且准备好一个 Query，你就做好了执行它并且将结果检索到内存中的准备。一次性将整个结果集检索到内存中是执行一个查询的最常用方式；我们将之称为列示。接下来我们还会探讨一些其他可用的选项，比如滚动和遍历。

14.3.1 列示所有结果

`getResultList()` 方法会执行 Query 并且将结果作为一个 `java.util.List` 返回：

```
Query query = em.createQuery("select i from Item i");
List<Item> items = query.getResultList();
```

Hibernate 会立即执行一个或几个 SQL SELECT 语句，这取决于你的抓取计划。如果用 `FetchType.EAGER` 映射任何关联或集合，那么除了你希望用查询检索的数据之外，Hibernate 还必须抓取它们。所有的数据都会被加载到内存中，且 Hibernate 检索的任何实体实例都处于持久化状态并由持久化上下文托管。

当然，持久化上下文不会管理标量投影结果。以下查询会返回 `String` 的 `List`：

```
Query query = em.createQuery("select i.name from Item i");
List<String> itemNames = query.getResultList();
```

使用一些查询，你就会知道其结果只是单一结果——例如，如果只想要最高 Bid 或只要一个 Item。

14.3.2 得到单个结果

可以使用 `getSingleResult()` 方法执行一个返回单个结果的查询：

```
TypedQuery<Item> query = em.createQuery(
    "select i from Item i where i.id = :id", Item.class
).setParameter("id", ITEM_ID);
```

```
Item item = query.getSingleResult();
```

调用 `getSingleResult()` 会返回一个 `Item` 实例。这也适用于标量结果：

```
TypedQuery<String> query = em.createQuery(
    "select i.name from Item i where i.id = :id", String.class
).setParameter("id", ITEM_ID);
```

```
String itemName = query.getSingleResult();
```

现在，不优雅的方面是：如果没有结果，则 `getSingleResult()` 会抛出一个 `NoResultException` 异常。这个查询会尝试找到一个具有不存在标识符的商品：

```
try {
    TypedQuery<Item> query = em.createQuery(
        "select i from Item i where i.id = :id", Item.class
    ).setParameter("id", 12341);

    Item item = query.getSingleResult();
    // ...
} catch (NoResultException ex) {
    // ...
}
```

你期望一个用于此类非常良性查询的 `null`。这简直是一个悲剧，因为它会强制你使用一个 `try/catch` 块保障此代码。实际上，它会强制你总是包装一个 `getSingleResult()` 调用，因为你无法知道是否会呈现这一(些)行。

如果有多个结果，则 `getSingleResult()` 会抛出一个 `NonUniqueResultException` 异常。使用这类查询通常会出现这种情况：

```
try {
    Query query = em.createQuery(
        "select i from Item i where name like '%a%'"
    );

    Item item = (Item) query.getSingleResult();
    // ...
} catch (NonUniqueResultException ex) {
    // ...
}
```

将所有结果检索到内存中是执行一个查询最常用的方式。如果希望优化一个查询的内存消耗以及执行行为，那么 `Hibernate` 支持你可能感兴趣的其他一些方法。

Hibernate 特性

14.3.3 滚动数据库游标

普通 `JDBC` 提供了一个被称为可滚动结果集的功能。此技术使用了数据库管理系统拥

有的一个游标。该游标指向查询结果中的特定行，并且应用程序可以向前和向后移动。甚至可以使用游标直接跳到一个行。

应该滚动查询结果而非将其全部加载到内存中的其中一种情形涉及太大而不适合加载到内存中的结果集。通常你会试图通过收紧查询中的条件来进一步限制结果。有时这是不可行的，可能是由于需要所有的数据却希望分几个步骤来检索它。我们将在 20.1 节中介绍这样一个批处理例程。

JPA 没有标准化使用数据库游标对结果进行滚动，因此需要 `org.hibernate.ScrollableResults` 接口在专有的 `org.hibernate.Query` 上可用：

```
Session session = em.unwrap(Session.class);

org.hibernate.Query query = session.createQuery( ← ①创建查询
    "select i from Item i order by i.id asc"
);
                                                    ②打开游标
org.hibernate.ScrollableResults cursor = ←
    query.scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

cursor.setRowNumber(2); ← ③跳到第三行
Item item = (Item) cursor.get(0); ← ④获得列值

cursor.close(); ← ⑤关闭游标
```

首先创建一个 `org.hibernate.Query` ① 并且打开一个游标 ②。然后忽略前两个结果行，跳到第三行 ③，并且得到该行的第一个“列”值 ④。JPQL 中没有列，因此这是第一个投影元素：即这里 `select` 子句中的 `i`。下一章中会介绍投影的更多示例。在结束数据库事务之前应该总是关闭该游标 ⑤！

可以从使用 `CriteriaBuilder` 构造的常规 `javax.persistence.Query` 中 `unwrap()` 该 Hibernate 查询 API。也可以用滚动替代 `list()` 执行专有的 `org.hibernate.Criteria` 查询；所返回的 `ScrollableResults` 游标也能起到相同作用。

Hibernate API 的 `ScrollMode` 常量相当于普通 JDBC 中的常量。在之前的示例中，`ScrollMode.SCROLL_INSENSITIVE` 意味着游标对于在数据库中做出的变更不敏感，以有效确保没有脏读取、不可重复的读取或者幻象读可以在滚动时出现在结果集中。其他可用的模式是 `SCROLL_SENSITIVE` 和 `FORWARD_ONLY`。敏感的游标会在其打开时将你暴露给已提交已修改的数据；并且使用一个仅向前的游标，无法跳到结果中的一个绝对位置。注意，即便是使用一个敏感游标，Hibernate 持久化上下文缓存仍然提供了对于实体实例的可重复读取，因此此设置仅可以影响你在结果集中投影的修改过的标量值。

注意，有些 JDBC 驱动不完全支持滚动数据库游标，尽管它看上去可能可行。例如，使用 MySQL 驱动，该驱动总是会立即将查询的整个结果集检索到内存中；因此你仅是在应用程序内存中滚动该结果集。要得到结果的真正逐行流，就必须将查询的 JDBC 抓取大小设置为 `Integer.MIN_VALUE` 并且仅使用 `ScrollMode.FORWARD_ONLY`。在使用游标之前，检查 DBMS 和 JDBC 驱动的行为和文档。

滚动数据库游标的一个重要限制是，它无法与使用 JPQL 中 `join fetch` 子句的动态抓取结合使用。联结抓取每次可能会处理几行，因此你不能逐行检索数据。如果尝试用动态抓

取子句对一个查询执行 `scroll()` 操作, 则 Hibernate 会抛出一个异常。

每次检索所有数据的另一个替代项是遍历。

Hibernate 特性

14.3.4 遍历一个结果

查询将检索的大多数实体实例目前都已经位于内存中。它们可能位于持久化上下文或者二级共享缓存中。在这样的情况下, 使用专有的 `org.hibernate.Query` API 遍历查询结果可能是合理的:

```
Session session = em.unwrap(Session.class);

org.hibernate.Query query = session.createQuery(
    "select i from Item i"
);

Iterator<Item> it = query.iterate(); // select ID from ITEM
while (it.hasNext()) {
    Item next = it.next(); // select * from ITEM where ID = ?
    // ...
}
Hibernate.close(it);
```

必须关闭 Iterator, 要么在关闭 Session 时关闭, 要么手动关闭

当调用 `query.iterate()` 时, Hibernate 会执行查询并且将一个 SQL SELECT 发送到数据库。但 Hibernate 会轻微修改该查询并且, 相较于检索 ITEM 表的所有列, 仅仅检索标识符/主键值。

然后, 每一次在 Iterator 上调用 `next()` 时, 会触发一个额外的 SQL 查询并且加载 ITEM 行的其余内容。显然, 这会造成 $n+1$ 查询问题, 除非 Hibernate 可以避免 `next()` 上的额外查询。如果 Hibernate 可以在持久化上下文缓存或者二级缓存中找到该商品的数据, 就会出现这种情况。

必须关闭由 `iterate()` 返回的 Iterator。Hibernate 会在关闭 EntityManager 或 Session 时自动关闭它。如果遍历过程超出了数据库中打开游标的最大数量, 则可以使用 `Hibernate.close(iterator)` 手动关闭 Iterator。

就该示例中所有拍卖商品必须位于缓存中以便让这个例程良好执行而言, 遍历的用处不大。就像滚动一个游标一样, 不能将它与动态抓取和 `join fetch` 子句结合使用; 如果尝试这么做, 则 Hibernate 将抛出一个异常。

到目前为止, 代码示例使用的全部都是 Java 代码中的嵌入查询字符串文字。这对于简单查询并不合理, 但是当开始考虑使用必须分成多行的复杂查询时, 它就有点不实用了。相反, 可以为每个查询提供一个名称并且将它移动到注解或 XML 文件中。

14.4 命名和外部化查询

外部化查询字符串让你可以存储所有与特定持久化类(或一组类)有关的查询以及用于该类的其他元数据。另外,可以将查询绑定到一个 XML 文件中,独立于任何 Java 类。较大的应用程序中通常会选用此技术;相较于将代码库分散在访问数据库的各个类中,在一些众所周知的位置维护数百个查询会更容易。要根据其名称引用和访问一个外部化的查询。

14.4.1 调用一个命名查询

`EntityManager#getNamedQuery()`方法会为一个命名查询获取一个 `Query` 实例:

```
Query query = em.createNamedQuery("findItems");
```

还可以为一个命名查询获取 `TypedQuery` 实例:

```
TypedQuery<Item> query = em.createNamedQuery("findItemById", Item.class);
```

Hibernate 的查询 API 也支持访问命名查询:

```
org.hibernate.Query query = session.getNamedQuery("findItems");
```

命名查询是全局的——即,一个查询的名称是特定持久化单元或 `org.hibernate.SessionFactory` 的唯一标识符。如何以及在 XML 文件或注解中的何处定义它们与应用程序代码无关。在启动时, Hibernate 会从 XML 文件和/或注解中加载命名 JPQL 查询并且解析它们以验证它们的语法(这在开发期间是有用的,但你可能希望在生产环境中禁用这个验证,对于较快的启动引导,可以使用持久化单元配置属性 `hibernate.query.startup_check`)。

即便是你用来编写命名查询的查询语言也无关紧要。它可以是 JPQL 或 SQL。

14.4.2 在 XML 元数据中定义查询

可以在 `orm.xml` 元数据的任何 JPA `<entity-mappings>` 元素中放置一个命名查询。在较大的应用程序中,我们建议将所有的命名查询隔离和分隔到其自己的文件中。另外,你可能想要相同的 XML 映射文件来定义查询和一个特定类。

`<named-query>` 元素会定义一个命名的 JPQL 查询:

路径: `/model/src/main/resources/querying/ExternalizedQueries.xml`

```
<entity-mappings
  version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

  <named-query name="findItems">
    <query><![CDATA[
```



```

        select i from Item i
    ]]></query>
</named-query>

</entity-mappings>

```

应该将查询文本包装到一个 CDATA 指令中,这样查询字符串中的任何可能不经意被视作 XML 的字符(比如小于运算符)就不会让 XML 解析器混淆。为了清晰可见,我们在大多数其他示例中省略了 CDATA。

命名查询不必在 JPQL 中编写。它们可以是原生 SQL 查询:

路径: /model/src/main/resources/querying/ExternalizedQueries.xml

```

<named-native-query name="findItemsSQL"
                    result-class="org.jpwh.model.querying.Item">
    <query>select * from ITEM</query>
</named-native-query>

```

如果认为稍后可能希望通过调优 SQL 来优化查询,那么这就是有用的。如果必须将一个遗留应用程序移植到 JPA/Hibernate,那么它也是一个好的解决方案,其中 SQL 代码可以与手动编码的 JDBC 例程分离。使用命名查询,可以轻易地逐个移植查询以映射文件。

Hibernate 特性

Hibernate 也有其自己的、非标准的设施,以用于在 Hibernate XML 元数据文件中外部化查询:

路径: /model/src/main/resources/querying/ExternalizedQueries.hbm.xml

```

<?xml version="1.0"?>
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <query name="findItemsOrderByAuctionEndHibernate">
        select i from Item i order by i.auctionEndasc
    </query>

    <sql-query name="findItemsSQLHibernate">
        <return class="org.jpwh.model.querying.Item"/>
        select * from ITEM order by NAME asc
    </sql-query>

</hibernate-mapping>

```

稍后将在第 17 章中看到外部化、尤其是自定义 SQL 查询的更多示例。

如果不喜欢 XML 文件,那么可以在 Java 注解元数据中绑定和命名查询。

14.4.3 使用注解定义查询

JPA 支持使用 @NamedQuery 和 @NamedNativeQuery 注解命名的查询。你只能在一个映射的类上放置这些注解。注意,查询名称也必须在所有的类中是全局唯一的;没有类或

包名称会被自动作为前缀加到查询名称上:

```
@NamedQueries({
    @NamedQuery(
        name = "findItemById",
        query = "select i from Item i where i.id = :id"
    )
})
@Entity
public class Item {
    // ...
}
```

此类是使用包含一组 `@NamedQuery` 的 `@NamedQueries` 来注解的。可以直接声明单个查询；不需要将它包装在 `@NamedQuery` 中。如果有一个 SQL 而非 JPQL 查询，则可以使用 `@NamedNativeQuery` 注解。有许多选项可以被考虑用于映射 SQL 结果集，因此稍后我们将在第 17 章专门的一节中介绍这如何生效。

Hibernate 特性

遗憾的是，JPA 的命名查询注解仅在它们位于一个映射类上时才有效。你不能将它们放到一个 `package-info.java` 元数据文件中。Hibernate 有一些用于该目的的专有注解：

路径：/model/src/main/java/org/jpwh/model/querying/package-info.java

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByName",
        query = "select i from Item i order by i.name asc"
    )
})

package org.jpwh.model.querying;
```

如果 XML 文件或注解看起来都不适合于定义你的命名查询，那么你可能希望编程式构造它们。

14.4.4 程式定义命名查询

使用 `EntityManagerFactory#addNamedQuery()` 方法将一个 Query “保存” 为一个命名查询：

```
Query findItemsQuery = em.createQuery("select i from Item i");
em.getEntityManagerFactory().addNamedQuery(
    "savedFindItemsQuery", findItemsQuery
);

Query query =
    em.createNamedQuery("savedFindItemsQuery");
```

稍后，使用相同的
EntityManagerFactory

这会将查询注册到持久化单元 `EntityManagerFactory`，并且让它可作为一个命名查询来重用。所保存的 `Query` 不必是一个 JPQL 语句；还可以保存一个条件或原生 SQL 查询。通常，要在应用程序启动时注册查询一次。

我们留待你来确定是否希望使用该命名查询功能。但我们将应用程序代码中的查询字符串(除非它们位于注解中)视作第二个选择；如果可能，应该总是外部化查询字符串。实际上，XML 文件可能是最通用的选项。

最后，对于有些查询，可能需要额外的设置和提示。

14.5 查询提示

在这一节中，我们将介绍来自 JPA 标准的一些额外查询选项以及一些专有的 Hibernate 设置。正如其名称所表明的，你可能不会立即需要它们，因此，如果愿意，可以跳过这一节，并且稍后作为参考来阅读它。

所有的示例都使用此处显示的相同查询：

```
String queryString = "select i from Item i";
```

一般来说，可以在一个 `Query` 上使用 `setHint()` 方法设置一个提示。所有其他的查询 API，比如 `TypedQuery` 和 `StoredProcedureQuery`，也具有这个方法。如果持久化提供程序不支持提示，则该提供程序将静默地忽略它。

JPA 标准化了表 14-1 中所示的提示。

表 14-1 标准化的 JPA 查询提示

名 称	值	描 述
<code>javax.persistence.query.timeout</code>	(毫秒)	为查询执行设置超时时长。这个提示在 <code>org.hibernate.annotations.QueryHints.TIMEOUT_</code> JPA 上也可以作为一个常量来使用
<code>javax.persistence.cache.retrieveMode</code>	<code>USE BYPASS</code>	控制在整理查询结果时 <code>Hibernate</code> 是否尝试从二级共享缓存中读取数据或者绕过缓存并且仅从查询结果中读取数据
<code>javax.persistence.cache.storeMode</code>	<code>USE BYPASS REFRESH</code>	控制在整理查询结果时 <code>Hibernate</code> 是否将数据存储在二级共享缓存中

Hibernate 特性

Hibernate 有其自己专用于查询的特定于供应商的提示，也可用作 `org.hibernate.annotations.QueryHints` 上的常量；参见表 14-2。

表 14-2 Hibernate 查询提示

名 称	值	描 述
org.hibernate.flushMode	org.hibernate.FlushMode(枚举)	控制在查询执行前，持久化上下文是否且何时应该被刷新
org.hibernate.readOnly	true false	为由查询返回的托管实体实例启用或禁用脏检查
org.hibernate.fetchSize	(JDBC 抓取大小)	在执行查询前调用 JDBC PreparedStatement#setFetchSize()方法，一个用于数据库驱动的优化提示
org.hibernate.comment	(SQL 注释字符串)	追加到 SQL 前的一个注释，用于(数据库)日志

二级共享缓存(尤其查询缓存)是一个复杂问题，因此我们将在 20.2 节专门探讨它。应该在启用该共享缓存之前阅读那一节：为一个查询将缓存设置为“启用”将没有任何效果。

其他一些提示也值得更详细的阐释。

14.5.1 设置一个超时时长

可以通过设置一个超时时长来控制让一个查询运行多长时间：

```
Query query = em.createQuery(queryString)
.setHint("javax.persistence.query.timeout", 60000); ◀—— 1 分钟
```

使用 Hibernate，这个方法就具有与 JDBC Statement API 上的 setQueryTimeout()方法相同的语义和结果。

注意，JDBC 驱动不必正好在出现超时时取消该查询。JDBC 规范规定，“一旦数据源有机会处理该请求来终止运行的命令，SQLException 将被抛出到客户端……。”因此，有必要解释何时才是数据源有机会终止命令的最佳时机。可能仅仅在执行完成后才是合适的时机。你可能希望用你的 DBMS 产品和驱动来验证这一点。

还可以指定此超时提示作为 persistence.xml 中的一个全局默认属性用作创建 EntityManagerFactory 时的属性或者用作一个命名查询选项。然后 Query#setHint()方法会为特定查询重写这个全局默认设置。

14.5.2 设置刷新模式

我们假设你在执行一个查询前对持久化实体实例做出了修改。例如，你要修改托管 Item 实例的名称。这些修改仅呈现在内存中，因此 Hibernate 默认会在执行你的查询前将该持久化上下文和所有变更刷新到数据库。这会保证查询运行在当前数据上并且查询结果和内存

实例之间不会出现冲突。

如果执行一个由许多查询-修改-查询-修改操作构成的系列操作，并且每个查询检索的数据集都与之前一个查询不同，那么这有时可能就是不切实际的。换句话说，有时候你知道在执行一个查询之前不需要将你的修改刷新到数据库，因为不一致的结果并不是一个问题。注意，持久化上下文为实体实例提供了可重复的读取，因此无论如何只有查询的标量结果才是一个问题。

可以在具有 Query 上的 `org.hibernate.flushMode` 提示的查询以及值 `org.hibernate.FlushMode.COMMIT` 之前禁用持久化上下文的刷新。幸运的是，JPA 在 `EntityManager` 和 Query API 上有一个标准的 `setFlushMode()` 方法，并且 `FlushModeType.COMMIT` 也是标准的。因此，如果希望仅在特定查询前禁用刷新，则可以使用标准 API：

```
Query query = em.createQuery(queryString)
    .setFlushMode(FlushModeType.COMMIT);
```

将刷新模式设置为 `COMMIT`，Hibernate 就不会在执行查询前刷新持久化上下文。默认设置为 `AUTO`。

14.5.3 设置只读模式

在 10.2.8 节中，我们探讨过你如何才能降低内存消耗并且避免长的脏检查周期。可以使用一个提示告知 Hibernate，它应该将由查询返回的所有实体实例视作只读(尽管未分离)：

```
Query query = em.createQuery(queryString).setHint(
    org.hibernate.annotations.QueryHints.READ_ONLY,true
);
```

所有由这个查询返回的 Item 实例都处于持久化状态，但 Hibernate 不会在持久化上下文中为自动脏检查启用快照。Hibernate 不会自动持久化任何修改，除非你使用 `session.setReadOnly(item, false)` 禁用只读模式。

14.5.4 设置一个抓取大小

抓取大小是用于数据库驱动的一个优化提示：

```
Query query = em.createQuery(queryString).setHint(
    org.hibernate.annotations.QueryHints.FETCH_SIZE,50
);
```

如果驱动未实现此功能，那么这个提示可能不会带来任何性能改进。如果它实现了此功能，那么当客户端(Hibernate)在一个查询结果上(即 `ResultSet` 上)进行操作时，就能通过在一个批次检索许多行来提升 JDBC 客户端和数据库之间的通信。

14.5.5 设置一个 SQL 注释

当优化一个应用程序时，通常必须阅读复杂的 SQL 日志。我们强烈建议在 `persistence.`

xml 配置中启用属性 `hibernate.use_sql_comments`。然后 Hibernate 会将一个自动生成的注释添加到它写入到日志的每个 SQL 语句。

可以用一个提示为特定 Query 设置自定义注释：

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.COMMENT,
        "Custom SQL comment"
    );
```

到目前为止你已经设置的提示都与 Hibernate 或 JDBC 处理有关。许多开发人员(和 DBA)都将查询提示视为完全不同的一些东西。在 SQL 中，查询提示是 SQL 语句中用于 DBMS 优化器的一个指令。例如，如果开发人员或 DBA 认为由数据库优化器选择的用于特定 SQL 语句的执行计划不是最快的，那么他们可以使用一个提示来强制执行一个不同的执行计划。Hibernate 和 Java 持久化不支持具有一个 API 的任意 SQL 提示；你将不得不回退到原生 SQL 并且编写你自己的 SQL 语句——当然可以使用提供的 API 执行该语句。

另一方面，使用一些 DBMS 产品，可以使用一个 SQL 注释在 SQL 语句的开头控制优化器。在这种情况下，要像最后一个示例中所示的那样使用该注释。

在之前所有的示例中，你已经在 Query 实例上直接设置了查询提示。如果已经对查询外部化并且命名了，则必须在注解或 XML 中设置提示。

14.5.6 命名的查询提示

之前使用 `setHint()` 设置的所有查询提示也可以在 XML 元数据的 `<named-query>` 或 `<named-native-query>` 元素中设置：

```
<entity-mappings
  version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

  <named-query name="findItems">
    <query><![CDATA[
      select i from Item i
    ]]></query>
    <hint name="javax.persistence.query.timeout" value="60000"/>
    <hint name="org.hibernate.comment" value="Custom SQL comment"/>
  </named-query>

</entity-mappings>
```

可以在注解中定义的命名查询上设置提示：


```

@NamedQueries({
    @NamedQuery(
        name = "findItemByName",
        query = "select i from Item i where i.name like :name",
        hints = {
            @QueryHint(
                name = org.hibernate.annotations.QueryHints.TIMEOUT_JPA,
                value = "60000"),
            @QueryHint(
                name = org.hibernate.annotations.QueryHints.COMMENT,
                value = "Custom SQL comment")
        }
    )
})

```

Hibernate 特性

可以在 package-info.java 文件的 Hibernate 注解中的命名查询上设置提示:

```

@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemBuyNowPriceGreaterThan",
        query = "select i from Item i where i.buyNowPrice > :price",
        timeout = 60,
        comment = "Custom SQL comment"
    )
})

```

← 秒!

```
package org.jpwh.model.querying;
```

当然,除此之外,也可以在外部化到 Hibernate XML 元数据文件中的命名查询上设置提示:

```

<?xml version="1.0"?>
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <query name="findItemsOrderByAuctionEndHibernateWithHints"
        cache-mode="ignore"
        comment="Custom SQL comment"
        fetch-size="50"
        read-only="true"
        timeout="60">
        select i from Item i order by i.auctionEndasc
    </query>

</hibernate-mapping>

```

14.6 本章小结

- 介绍了如何创建和执行查询。要创建查询,你要使用 JPA 查询接口并且处理类型化的查询结果。你还看到了 Hibernate 自己的查询接口。

查询语言

第 15 章

15

本章内容简介:

- 编写 JPQL 和条件查询
- 使用联结有效地检索数据
- 报告查询和子查询

查询是编写优秀数据访问代码的最有意思的部分。一个复杂的查询可能需要长时间准备，并且其对应用程序性能的影响可能是惊人的。另一方面，你的经验越丰富，编写查询就会越容易，且最初看起来复杂的查询只是与你是否了解可用的查询语言有关。

本章将介绍 JPA 中可用的查询语言：JPQL 和条件查询 API。我们会一直使用这两种语言/API 介绍相同的查询示例，其中查询结果是相等的。

JPA 2 中主要的新功能

- 现在有了对 CASE、NULLIF 和 COALESCE 操作符的支持，使用与其 SQL 相对物相同的语义。
- 可以在限制和选择中使用 TREAT 操作符进行向下类型转换。
- 可以在限制和投影中调用任意 SQL 数据库函数。
- 可以使用新的 ON 关键字为外联结附加额外的联结条件。
- 可以在子查询 FROM 子句中使用联结。

我们期望你不是仅仅阅读本章一次，而是将其作为参考，以便在为你的应用程序编码时为特定查询查找正确的语法。因此，本章所编写的内容不会太冗长，它具有用于不同用例的许多小的代码示例。为了更好的可读性，有时我们还会简化 CaveatEmptor 应用程序的各个部分。例如，相比于提及 MonetaryAmount，我们会使用一个简单 BigDecimal 金额作为比较。

我们首先介绍一些查询的专业术语。你要应用选择来定义应该从何处检索数据、限制对指定条件的记录匹配以及投影以便选择你想要从查询中返回的数据。你会发现本章正是以此方式来组织的。

当我们在本章中谈论查询时，我们通常指的是 SELECT 语句：从数据库检索数据的操作。JPA 也支持 JPQL、条件和 SQL 方式中的 UPDATE、DELETE 甚至 INSERT ...SELECT 语句，我们将在 20.1 节中探讨这些内容。我们不会在此处重复那些大量的操作，而是专注

于 SELECT 语句。我们从一些基本的选择示例开始介绍。

15.1 选择

首先，当我们谈及查询时，我们指的并非是查询的 SELECT 子句。我们也不是在谈论像 SELECT 语句这样的东西。我们指的是选择一个关系变量——或者，用 SQL 术语说就是 FROM 子句。它会声明用于查询的数据应该从哪里获取：简单来说，就是要为查询“选择”哪些表。或者，使用类而非 JPQL 中的表名称：

```
from Item
```

上述查询(仅一个 FROM 子句)会检索所有的 Item 实体实例。Hibernate 会生成以下 SQL：

```
select i.ID, i.NAME, ... from ITEM i
```

可以使用 from() 方法构建同等的条件查询，传入实体名称：

```
CriteriaQuery criteria = cb.createQuery(Item.class);
criteria.from(Item.class);
```

Hibernate 特性

Hibernate 理解只有一个 FROM 子句或条件的查询。遗憾的是，我们刚才介绍的 JPQL 和条件查询不是可移植的；它们并非 JPA 兼容。JPA 规范要求 JPQL 查询具有一个 SELECT 子句并且可移植的条件查询调用 select() 方法。

这就要求指定别名和查询根，这是我们接下来的主题。

15.1.1 指定别名和查询根

将一个 SELECT 子句添加到一个 JPQL 查询要求将一个别名指定到 FROM 子句中所查询的类，这样一来你才能在查询的其他部分引用它：

```
select i from Item as i
```

现在，上述查询是 JPA 兼容的。as 关键字总是可选的。以下查询是等效的：

```
select i from Item i
```

将 i 指定给所查询的 Item 类的实例。可以将其看成有点类似于以下 Java 代码中的临时变量声明：

```
for(Iterator i = result.iterator(); i.hasNext();) {
    Item item = (Item) i.next();
    // ...
}
```

查询中的别名不是大小写敏感的，因此 select iTm from Item itm 也可以运行。不过，我们更愿意保持别名简短和简单；它们只需要在一个查询(或子查询)中是唯一的即可。

可移植的条件查询必须调用 `select()` 方法：

```
CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);
criteria.select(i);
```

我们将在其他大多数条件示例中跳过 `cb.createQuery()` 行；它总是相同的。无论何时看到一个 `criteria` 变量，它都会由 `CriteriaBuilder#createQuery()` 生成。上一章阐释了如何获得一个 `CriteriaBuilder`。

条件查询的 `Root` 总是会引用一个实体。稍后我们将介绍具有几个根的查询。可以通过内联 `Root` 简化此查询：

```
criteria.select(criteria.from(Item.class));
```

或者，可以使用 `Metamodel` API 动态查找实体类型：

```
EntityType entityType = getEntityType(
    em.getMetamodel(), "Item"
);
criteria.select(criteria.from(entityType));
```

`getEntityType()` 方法是我们自己的一个小的附加方法：它会遍历 `Metamodel#getEntities()`，使用指定实体名称查找一个匹配。

`Item` 实体不具有子类，所以我们接下来看看多态查询。

15.1.2 多态查询

作为一种面向对象的查询语言，JPQL 支持多态查询——分别用于一个类的实例和其子类的所有实例的查询。思考以下查询：

```
select bd from BillingDetails bd
criteria.select(criteria.from(BillingDetails.class));
```

这些查询会返回 `BillingDetails` 类型的所有实例，它是一个抽象类。在这种情况下，每一个实例都是 `BillingDetails` 的一个子类型：`CreditCard` 或 `BankAccount`。如果只想要某个特定子类的实例，则可以使用这个：

```
select cc from CreditCard cc
criteria.select(criteria.from(CreditCard.class));
```

在 `FROM` 子句中命名的类甚至不需要是一个映射的持久化类；任何类都行。以下查询会返回所有的持久化对象：

```
select o from java.lang.Object o
```

Hibernate 特性

你的确可以使用这样一个查询选择数据库的所有表并且将所有的数据检索到内存中！

这也适用于任何接口——例如，查询所有可序列化类型：

```
select s from java.io.Serializable s
```

遗憾的是，JPA 并没有用任何接口标准化多态 JPQL 查询。它们适用于 Hibernate 中，但可移植的应用程序仅应该在 FROM 子句中引用映射的实体类(比如 BillingDetails 或 CreditCard)。条件查询 API 中的 from()方法只接受映射的实体类型。

可以通过使用 TYPE 函数限制所选择类型的范围来执行非多态查询。如果只想要某个特定子类的实例，则可以使用：

```
select bd from BillingDetails bd where type(bd) = CreditCard
```

```
Root<BillingDetails> bd = criteria.from(BillingDetails.class);
criteria.select(bd).where(
    cb.equal(bd.type(), CreditCard.class)
);
```

如果需要参数化这样一个查询，则可以添加一个 IN 子句和一个命名参数：

```
select bd from BillingDetails bd where type(bd) in :types
Root<BillingDetails> bd = criteria.from(BillingDetails.class);
criteria.select(bd).where(
    bd.type().in(cb.parameter(List.class, "types"))
);
```

要通过提供一个希望匹配的类型的 List 来将实参绑定到形参：

```
Query query = // ...
query.setParameter("types", Arrays.asList(CreditCard.class,
    BankAccount.class));
```

如果想要除了一个指定类之外的某个特定子类的所有实例，则可以使用以下查询：

```
select bd from BillingDetails bd where not type(bd) = BankAccount
Root<BillingDetails> bd = criteria.from(BillingDetails.class);
criteria.select(bd).where(
    cb.not(cb.equal(bd.type(), BankAccount.class))
);
```

多态性不仅应用在显式命名的类上，还会应用到多态关联，本章稍后将会介绍这一点。

现在你完成了编写一个查询的第一步，选择。你选取了从中查询数据的表。接下来，你可能希望用一个限制来约束你想要检索的行。

15.2 限制

通常，你不希望从数据库检索一个类的所有实例。你必须能够表达由查询返回的数据

上的约束。我们称之为限制。WHERE 子句声明了 SQL 和 JPQL 中的限制条件，而 where() 方法是条件查询 API 中的等同物。

这是一个典型的 WHERE 子句，它将结果限制为具有一个指定名称的所有 Item 实例：

```
select i from Item i where i.name = 'Foo'

Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(cb.equal(i.get("name"), "Foo"))
);
```

该查询根据 Item 类的 name 属性表述了约束。

由这些查询生成的 SQL 是：

```
select i.ID, i.NAME, ... from ITEM i where i.NAME = 'Foo'
```

可以使用单引号在你的语句和条件中包含字符串文字。对于日期、时间和时间戳文字，可以使用 JDBC 转义语法：...where i.auctionEnd = {d '2013-26-06'}。注意，JDBC 驱动和 DBMS 定义了如何解析此文字以及它们支持哪些其他的变体。记住我们上一章的建议：不要将未过滤的用户输入串联到查询字符串中——要使用参数绑定。JPQL 中其他常见的文字是 true 和 false：

```
select u from User u where u.activated = true

Root<User> u = criteria.from(User.class);
criteria.select(u).where(
    cb.equal(u.get("activated"), true)
);
```

SQL(JPQL 和条件查询)使用三元逻辑表述了限制。WHERE 子句是一个逻辑表达式，它的计算结果为 true、false 或 null。

什么是三元逻辑？

当且仅当 WHERE 子句的计算结果为 true 时，SQL 查询结果中才会包含一行。在 Java 中，nonNullObject == null 的计算结果为 false，而 null == null 的计算结果为 true。在 SQL 中，NOT_NULL_COLUMN = null 和 null = null 的计算结果都为 null，而非 true。因此，SQL 需要特殊的操作符 IS NULL 和 IS NOT NULL，以便验证一个值是否为 null。三元逻辑是处理可以应用到可为空列值的表达式的一种方式。不要将 null 当作一个特殊标记，而是作为一个普通值，它是大家熟知的关系模型三元逻辑的一个 SQL 扩展。Hibernate 必须用 JPQL 和条件查询中的三元操作符支持此三元逻辑。

我们来逐一看看逻辑表达式中最常用的比较操作符，其中包括三元操作符。

15.2.1 比较表达式

JPQL 和条件 API 支持与 SQL 相同的基础比较操作符。如果了解 SQL 的话，这里有一些你看起来应该很熟悉的示例。

以下查询会返回指定金额范围中的所有出价：

```
select b from Bid b where b.amount between 99 and 110
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.between(
        b.<BigDecimal>get("amount"),
        new BigDecimal("99"), new BigDecimal("110")
    )
);
```

所需的路径类型

必须是相同类型

条件查询可能看起来有点奇怪；可能在 Java 中还没有经常看到过表达式中间的泛型。Root#get()方法会生成实体属性的一个 Path<X>。要保持类型安全，必须指定 Path 的属性类型，就像<BigDecimal>get("amount")中一样。然后 between()方法的其他两个参数必须是相同类型，否则该比较就没什么意义或无法编译。

以下查询会返回具有比指定值更大金额的所有出价：

```
select b from Bid b where b.amount > 100
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.gt(
        b.<BigDecimal>get("amount"),
        new BigDecimal("100")
    )
);
```

gt()仅适用于 Number；否则就要使用 greaterThan()

gt()方法仅接受 Number 类型的参数，比如 BigDecimal 或 Integer。如果需要比较其他类型的值，比如一个 Date，则要转而使用 greaterThan():

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.greaterThan(
        i.<Date>get("auctionEnd"),
        tomorrowDate
    )
);
```

以下查询将返回所有具有用户名“johndoe”和“janeroe”的用户：

```
select u from User u where u.username in ('johndoe', 'janeroe')
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(
    cb.<String>in(u.<String>get("username"))
        .value("johndoe")
        .value("janeroe")
);
```

对于使用枚举的限制，则要使用完全限定的文字：

```
select i from Item i
    where i.auctionType = org.jpwh.model.querying.AuctionType.HIGHEST_BID
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.equal(
        i.<AuctionType>get("auctionType"),
        AuctionType.HIGHEST_BID
    )
);
```

由于 SQL 依赖三元逻辑，因此要验证 `null` 值需要特别注意。要使用 JPQL 中的 `IS [NOT] NULL` 操作符以及条件查询 API 中的 `isNotNull()`。

这里是 `IS NULL` 和 `isNull()` 在起作用，得到没有一口价的商品：

```
select i from Item i where i.buyNowPrice is null
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(cb.isNull(i.get("buyNowPrice")))
);
```

使用 `IS NOT NULL` 和 `isNotNull()`，就可以返回具有一口价的商品：

```
select i from Item i where i.buyNowPrice is not null
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(cb.isNotNull(i.get("buyNowPrice")))
);
```

`LIKE` 操作符允许通配符搜索，其中的通配符就是 `%` 和 `_`，就像在 SQL 中一样：

```
select u from User u where u.username like 'john%'
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(cb.like(u.<String>get("username"), "john%"))
);
```

表达式 `john%` 会将结果限制为具有以 “john” 开头的 `username` 的用户。例如，还可以在子字符串匹配表达式中取消 `LIKE` 操作符：

```
select u from User u where u.username not like 'john%'
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(cb.like(u.<String>get("username"),
"john%").not())
);
```

可以通过使用百分号字符将搜索字符串包围起来以便匹配任何子字符串：


```
select u from User u where u.username like '%oe%'
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(cb.like(u.<String>get("username"), "%oe%"))
);
```

百分号支持任何字符序列；下划线可被用来通配单个字符。如果想要一个文字上的百分号或下划线，则可以转义你选择的字符：

```
select i from Item i
    where i.name like 'Name\_with\_underscores' escape :escapeChar
query.setParameter("escapeChar", "\\");
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.like(i.<String>get("name"), "Name\\\_with\\\_underscores", '\\')
);
```

这些查询会返回所有具有 `Name_with_underscores` 的商品。在 Java 字符串中，字符是转义字符，因此你必须对它转义，这就解释了示例中的双反斜杠。

JPA 也支持算术表达式：

```
select b from Bid b where (b.amount / 2) - 0.5 > 49
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.gt(
        cb.diff(
            cb.quot(b.<BigDecimal>get("amount"), 2),
            0.5
        ),
        49
    )
);
```

逻辑操作符(以及用于分组的圆括号)合并了表达式：

```
select i from Item i
    where (i.name like 'Fo%' and i.buyNowPrice is not null)
        or i.name = 'Bar'/
```

```
Root<Item> i = criteria.from(Item.class);
```

```
Predicate predicate = cb.and(
    cb.like(i.<String>get("name"), "Fo%"),
    cb.isNotNull(i.get("buyNowPrice"))
);
```

```
predicate = cb.or(
    predicate,
    cb.equal(i.<String>get("name"), "Bar")
);
```

```
);

criteria.select(i).where(predicate);
```

Hibernate 特性

如果用逻辑 AND 合并所有的谓词，那么我们推荐以下流畅的条件查询 API 方式：

```
Root<Item> i = criteria.from(Item.class);

criteria.select(i).where(
    cb.like(i.<String>get("name"), "Fo%"),
    // AND
    cb.isNotNull(i.get("buyNowPrice"))
    // AND ...
);
```

我们在表 15-1 中从上到下汇总了所有的操作符及其优先级，包括到目前为止我们还没有介绍的一些。

表 15-1 JPQL 操作符优先级

JPQL 操作符	条件查询 API	描 述
.	N/A	导航路径表达式操作符
+, -	neg()	一元正或负号(所有无符号的数值都会被视作正值)
*, /	prod(), quot()	数值的乘和除
+, -	sum(), diff()	数值的加和减
=, <, <=, >, >=, <	equal(), notEqual(), lessThan(), lt(), greaterThan(), gt(), greaterThanEqual(), ge(), lessThan(), lt()	具有 SQL 语义的二元比较操作符
[NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL	between(), like(), in(), isNull(), isNotNull()	具有 SQL 语义的二元比较操作符
IS [NOT] EMPTY, [NOT]MEMBER [OF]	isEmpty(), isNotEmpty(), isMember(), isNotMember()	用于持久化集合的二元操作符
NOT, AND, OR	not(), and(), or()	用于表达式估算顺序的逻辑操作符

已经介绍了二元比较表达式如何具有与其 SQL 对等物相同的语义，以及如何使用逻辑操作符对它们进行分组与合并。我们现在探讨集合处理。

15.2.2 使用集合的表达式

前面几节中的所有表达式都只有单值的路径表达式：`user.username`、`item.buyNowPrice` 等。还可以编写以集合结尾的路径表达式，并且应用一些操作符和函数。

例如，我们将查询结果限制为 `Category` 实例，这些实例的 `items` 集合中有一个元素：

```
select c from Category c
    where c.items is not empty

Root<Category> c = criteria.from(Category.class);
criteria.select(c).where(cb.isNotEmpty(c.<Collection>get("items")))
);
```

JPQL 查询中的 `c.items` 路径表达式结束于一个集合属性：一个 `Category` 的 `items`。注意，在一个集合值属性之后继续使用路径表达式总是非法的：不能编写 `c.items.buyNowPrice`。

可以使用 `size()` 函数根据集合的大小限制结果：

```
select c from Category c
    where size(c.items) > 1

Root<Category> c = criteria.from(Category.class);
criteria.select(c).where(
    cb.gt(
        cb.size(c.<Collection>get("items")),
        1
    )
);
```

还可以表述需要集合中提供一个特定元素：

```
select c from Category c
    where :item member of c.items

Root<Category> c = criteria.from(Category.class);
criteria.select(c).where(
    cb.isMember(
        cb.parameter(Item.class, "item"),
        c.<Collection<Item>>get("items")
    )
);
```

对于持久化映射，可以使用特殊的操作符 `key()`、`value()` 和 `entry()`。我们假设每个 `Item` 都有一个可嵌入的 `Image` 的持久化映射，就像 7.2.4 节中所示一样。每个 `Image` 的文件名都是一个映射键。以下查询会检索文件名中具有 `.jpg` 后缀的所有 `Image` 实例：

```
select value(img)
    from Item i join i.images img
    where key(img) like '%.jpg'
```


value()操作符会返回 Map 的值，而 key()操作符会返回 Map 的键集。如果希望返回 Map.Entry 实例，则可以使用 entry()操作符。

我们接下来看看其他可用的函数，并不局限于集合。

15.2.3 调用函数

查询语言的一个非常强大的功能是在 WHERE 子句中调用函数的功能。以下查询调用了 lower()函数用于对大小写不敏感的搜索：

```
select i from Item i where lower(i.name) like 'ba%'

Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.like(cb.lower(i.<String>get("name")), "ba%")
);
```

看看表 15-2 中所有可用函数的汇总。对于条件查询，等效的方法是在 CriteriaBuilder 中，使用稍微不同的名称格式化(使用驼峰格式并且不使用下划线)。

表 15-2 JPA 查询函数(未列出重载的方法)

函 数	适 用 范 围
upper(s), lower(s)	字符串值；返回一个字符串值
concat(s, s)	字符串值；返回一个字符串值
current_date, current_time, current_timestamp	返回数据库管理系统机器的日期和/或时间
substring(s, offset, length)	字符串值(偏移从 1 开始)；返回一个字符串值
trim([[both leading trailing] char [from]] s)	如果未指定 char 或其他规范，则清除两端的空格；返回一个字符串值
length(s)	字符串值；返回一个数值
locate(search, s, offset)	返回从 s 中的 offset 开始搜索的 search 的位置；返回一个数值
abs(n), sqrt(n), mod(dividend, divisor)	数值；返回与输入类型相同的绝对值、类型为 Double 的平方根，以及类型为 Integer 的除法余数
treat(x as Type)	在限制中向下类型转换；例如，检索信用卡于 2013 年过期的所有用户：select u from User u where treat(u.billingDetails as CreditCard).expYear = '2013'(注意这在 Hibernate 中不是必须的。如果使用了一个子类属性路径，则会自动向下类型转换)
size(c)	集合表达式；返回一个 Integer，或者如果为空，则返回 0

(续表)

函 数	适 用 范 围
index(orderedCollection)	用于使用@OrderColumn 映射的集合的表达式；返回与列表中其参数位置对应的一个 Integer 值。例如，select i.name from Category c join c.items i where index(i) = 0 会返回每个类别中第一个商品的名称

Hibernate 特性

Hibernate 为 JPQL 提供了额外的函数，如表 15-3 所示。标准 JPA 条件 API 中没有这些函数的对等物。

表 15-3 Hibernate 查询函数

函 数	描 述
bit_length(s)	返回 s 中的位数
second(d), minute(d), hour(d), day(d), month(d), year(d)	从时态参数中提取时间和日期
minelement(c), maxelement(c), minindex(c), maxindex(c), elements(c), indices(c)	返回一个索引过的集合(映射、列表、数组)的元素或索引
str(x)	将参数转换成字符串

大多数这些仅用于 Hibernate 的函数都可以转换成 SQL 对等物。还可以调用 DBMS 支持的此处未列出的 SQL 函数。

使用 Hibernate,任何 Hibernate 不知道的在 JPQL 语句的 WHERE 子句中调用的函数都会作为一个 SQL 函数调用被直接传递给数据库。例如，以下查询会返回拍卖周期大于一天的所有商品：

```
select i from Item i
where
    datediff('DAY', i.createdOn, i.auctionEnd)
> 1
```

此处你调用了 H2 数据库系统的专有 datediff()函数，它会返回 Item 的创建日期和拍卖结束日期之间的天数差别。不过，这个语法仅适用于 Hibernate 中；在 JPA 中，用于调用任意 SQL 函数的标准调用语法是：

```
select i from Item i
where
    function('DATEDIFF', 'DAY', i.createdOn, i.auctionEnd)
> 1
```

function()的第一个参数是你希望在单引号中调用的 SQL 函数的名称。然后，要为实际

的函数附加任何额外的运算对象；可以一个都不使用或者使用多个。这是相同的条件查询：

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.gt(
        cb.function(
            "DATEDIFF",
            Integer.class,
            cb.literal("DAY"),
            i.get("createdOn"),
            i.get("auctionEnd")
        ),
        1
    )
);
```

`Integer.class` 参数是 `datediff()` 函数的返回类型，并且此处是不相关的，因为并不是在返回限制中调用的函数的结果。

`SELECT` 子句中的函数调用会将值返回到 Java 层；还可以调用 `SELECT` 子句中的任意 SQL 数据库函数。在我们讨论此子句和投影之前，先看看如何才能对结果排序。

15.2.4 对查询结果排序

所有的查询语言都提供了用于对查询结果排序的一些机制。JPQL 提供了一个 `ORDER BY` 子句，类似于 SQL。

以下查询会返回所有的用户，根据用户名进行排序，默认为升序：

```
select u from User u order by u.username
```

要用 `asc` 和 `desc` 关键字指定升序或降序：

```
select u from User u order by u.username desc
```

使用条件查询 API，必须用 `asc()` 或 `desc()` 指定升序和降序：

```
Root<User> u = criteria.from(User.class);
criteria.select(u).orderBy(
    cb.desc(u.get("username"))
);
```

可以根据多个属性进行排序：

```
select u from User u order by u.activated desc, u.username asc
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).orderBy(
    cb.desc(u.get("activated")),
    cb.asc(u.get("username"))
);
```


null 的顺序

如果正在排序的列可以是 NULL，那么具有 NULL 的行就可能位于查询结果的最前或者最后。此行为取决于 DBMS，因此对于可移植的应用程序，应该使用子句 ORDER BY... NULLS FIRST|LAST 指定 NULL 应该位于最前还是最后。Hibernate 支持 JPQL 中的这个子句，不过，这在 JPA 中并非是标准化的。或者，可以将持久化单元配置属性 hibernate.order_by.default_null_ordering 设置为 none(默认设置)、first 或 last 来设置一个默认顺序。

Hibernate 特性

如果 SELECT 子句投影相同的属性/路径，则 JPA 规范只允许在 ORDER BY 子句中使用属性/路径。以下查询可能是不可移植的，但在 Hibernate 中可用：

```
select i.name from Item i order by i.buyNowPrice asc
select i from Item i order by i.seller.username desc
```

要小心路径表达式和 ORDER BY 中的隐式内联结：最后一个查询只返回具有一个 seller 的 Item 实例。这可能并非是预期的，正如不具有 ORDER BY 子句的相同查询会检索所有 Item 实例一样(暂时忽略模型中 Item 总是有一个 seller 的情况，这个问题在使用可选引用时会出现)。本章稍后将更为详尽地探讨内联结和路径表达式。

你现在知道了如何编写 FROM、WHERE 以及 ORDER BY 子句。了解了如何选择你希望检索其实例的实体以及必要的表达式和操作来限制结果并且对其排序。目前你所需要的就是将这个结果的数据投影到应用程序中所需内容的功能。

15.3 投影

简单来说，一个查询中的选择和限制就是声明你希望查询哪些表和行的过程。投影就是定义你想要返回给应用程序的“列”：需要的数据。JPQL 中的 SELECT 子句会执行投影。

15.3.1 实体和标量值的投影

例如，思考以下查询：

```
select i, b from Item i, Bid b
```

```
Root<Item> i = criteria.from(Item.class);
Root<Bid> b = criteria.from(Bid.class);
criteria.select(cb.tuple(i, b));
/* Convenient alternative:
criteria.multiselect(
    criteria.from(Item.class),
    criteria.from(Bid.class)
);
*/
```

如前所述, 这个条件查询显示了如何才能通过几次调用 `from()` 方法来添加几个 `Root`。要将几个元素添加到投影, 要么可以调用 `CriteriaBuilder` 的 `tuple()` 方法, 要么可以调用简化的 `multiselect()`。

正在创建所有 `Item` 和 `Bid` 实例的笛卡尔积。查询会返回排序过的 `Item` 和 `Bid` 实体实例对:

```
List<Object[]> result = query.getResultList(); ← ①返回 Object[] 的 List
```

```
Set<Item> items = new HashSet();
```

```
Set<Bid> bids = new HashSet();
```

```
for (Object[] row : result) {
    assertTrue(row[0] instanceof Item); ← ②索引 0
    items.add((Item) row[0]);
```

```
    assertTrue(row[1] instanceof Bid); ← ③索引 1
    bids.add((Bid) row[1]);
```

```
}
```

```
assertEquals(items.size(), 3);
```

```
assertEquals(bids.size(), 4);
```

```
assertEquals(result.size(), 12); ← ④笛卡尔积
```

该查询会返回 `Object[]` 的一个 `List`①。索引 0 是 `Item`②, 索引 1 是 `Bid`③。

由于这是一个积, 因此该结果包含这两个底层表中可以找到的 `Item` 和 `Bid` 行的每一个可能的组合。显然, 这个查询没什么用, 但你应该不会对将 `Object[]` 的集合作为查询结果来检索感到惊讶。Hibernate 会在持久化上下文中管理所有处于持久化状态的 `Item` 和 `Bid` 实体实例。注意 `HashSets` 如何过滤掉重复的 `Item` 和 `Bid` 实例。

或者, 使用 `Tuple` API, 在条件查询中得到对结果列表的类型化访问。首先调用 `createTupleQuery()` 创建一个 `CriteriaQuery<Tuple>`。然后, 通过为实体类添加别名来改进查询定义:

```
CriteriaQuery<Tuple> criteria = cb.createTupleQuery();
```

```
// Or: CriteriaQuery<Tuple> criteria = cb.createQuery(Tuple.class);
```

```
criteria.multiselect(
    criteria.from(Item.class).alias("i"), ← 别名是可选的
    criteria.from(Bid.class).alias("b")
);
```

```
TypedQuery<Tuple> query = em.createQuery(criteria);
```

```
List<Tuple> result = query.getResultList();
```

`Tuple` API 提供了几种方式来访问结果, 根据索引、根据别名, 或者非类型化的元访问:

```
for (Tuple tuple : result) {
    Item item = tuple.get(0, Item.class); ← 索引过的
    Bid bid = tuple.get(1, Bid.class);
```

```

item = tuple.get("i", Item.class);
bid = tuple.get("b", Bid.class);
for (TupleElement<?> element : tuple.getElements()) {
    Class clazz = element.getJavaType();
    String alias = element.getAlias();
    Object value = tuple.get(element);
}

```

← 别名 ← 元

以下投影也会返回 `Object[]` 的一个集合:

```

select u.id, u.username, u.homeAddress from User u
Root<User> u = criteria.from(User.class);
criteria.multiselect(u.get("id"), u.get("username"), u.get("homeAddress"));

```

返回 `Object[]` 的一个 List

由此查询返回的 `Object[]` 包含索引 0 位置的一个 `Long`、索引 1 位置的一个 `String` 以及索引 2 位置的一个 `Address`。前两个是标量值；第三个是一个嵌入的类实例。它们都不是托管的实体实例！因此，这些值不能像实体实例那样处于任何持久化状态。它们并非事务性的，并且显然不能被自动检查脏状态。我们称所有这些值都是瞬时的。这是需要为简单报告界面编写的查询种类，以便显示所有的用户名称及其家庭地址。

你现在已经几次看到路径表达式了：使用圆点符号，可以引用一个实体的属性，比如使用 `u.username` 引用 `User#username`。例如，对于一个嵌套的嵌入属性，可以编写路径 `u.homeAddress.city.zipcode`。这些都是单一值的路径表达式，因为它们没有结束于一个映射的集合属性。

相较于 `Object[]` 或 `Tuple`，尤其是对于报告查询来说，一个更便利的替代项是投影中的动态实例化，这是接下来的主题。

15.3.2 使用动态实例化

我们假设你的应用程序中有一个报告界面，其中需要显示一个列表中的一些数据。你希望显示所有的拍卖商品以及每次拍卖的结束时间。你不希望加载托管的 `Item` 实体实例，因为没有数据会被修改：你只读取数据。

首先，编写一个称为 `ItemSummary` 的具有一个构造函数的类，该构造函数要使用一个 `Long` 用于商品的标识符、一个 `String` 用于商品的名称以及一个 `Date` 用于商品的拍卖结束时间戳：

```

public class ItemSummary {
    public ItemSummary(Long itemId, String name, Date auctionEnd) {
        // ...
    }
    // ...
}

```


有时我们将这些类型的类称为数据传输对象(Data Transfer Objects, DTO), 因为它们的主要目的是在应用程序中来回传递数据。`ItemSummary` 类没有被映射到数据库, 并且可以根据报告用户界面的需要添加任意方法(获取方法、设置方法、值打印)。

Hibernate 可以使用 JPQL 中的 `new` 关键字和条件中的 `construct()` 方法直接从一个查询中返回 `ItemSummary` 的实例:

```
select new org.jpwh.model.querying.ItemSummary(
    i.id, i.name, i.auctionEnd
) from Item i
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(
    cb.construct(
        ItemSummary.class,
        i.get("id"), i.get("name"), i.get("auctionEnd")
    )
);
```

必须具有正确的构造函数

在这个查询的结果列表中, 每个元素都是 `ItemSummary` 的一个实例。注意在 JPQL 中, 必须使用一个完全限定的类名称, 这意味着包含包名称。还要注意, 不支持嵌套构造函数调用: 不能编写 `new ItemSummary(..., new UserSummary(...))`。

动态实例化并不局限于像 `ItemSummary` 这样的非持久化数据传输类。可以在查询中构造一个新 `Item` 或 `User`, 它是一个映射的实体类。唯一重要的规则是, 这个类必须具有一个匹配构造函数用于投影。但如果动态构造实体实例, 那么在从查询返回时它们就不会处于持久化状态! 它们将处于瞬时状态或分离状态, 这取决于是否设置标识符值。此功能的一个用例是简单数据复制: 从数据库检索具有一些复制到构造函数中的值的一个“新”瞬时 `Item`, 在应用程序中设置其他一些值, 然后使用 `persist()` 将它存储到数据库中。

如果 DTO 类不具有正确的构造函数, 并且希望通过设置方法或者字段从一个查询结果中填充它, 则要应用一个 `ResultTransformer`, 如 16.1.3 节中所示。稍后将介绍聚合与分组的更多示例。

接下来, 我们要查看使用投影的一个问题, 它对于许多工程师来说常常都难以理解: 处理重复项。

15.3.3 得到唯一结果

当在一个查询中创建投影时, 无法保证其结果的元素是唯一的。例如, 商品名称不是唯一的, 因此以下查询可能会多次返回相同的名称:

```
select i.name from Item i
```

```
CriteriaQuery<String> criteria = cb.createQuery(String.class);
criteria.select(
    criteria.from(Item.class).<String>get("name")
);
```

弄明白一个查询结果中具有两个完全相同的行有什么意义是很难的，因此如果认为有可能出现重复，那么通常要应用 **DISTINCT** 关键字或 `distinct()` 方法：

```
select distinct i.name from Item i

CriteriaQuery<String> criteria = cb.createQuery(String.class);
criteria.select(
    criteria.from(Item.class).<String>get("name")
);
criteria.distinct(true);
```

这样就能从 `Item` 描述的返回列表中消除重复项并且直接转换成 SQL **DISTINCT** 操作符。该过滤发生在数据库级别。本章稍后的内容中，你会看到事实的情况并非总是如此。

之前，你看到了限制的 **WHERE** 子句中的函数调用。还可以在投影中调用函数，以便在查询内修改返回的数据。

15.3.4 在投影中调用函数

以下查询在投影中使用 `concat()` 函数返回了一个自定义字符串：

```
select concat(concat(i.name, ': '), i.auctionEnd) from Item i

Root<Item> i = criteria.from(Item.class);
criteria.select(
    cb.concat(
        cb.concat(i.<String>get("name"), ":"),
        i.<String>get("auctionEnd")
    )
);
```

← 注意 Date 的转换。

此查询会返回 `String` 的一个 `List`，每个 `String` 都具有 “[商品名称]:[拍卖结束日期]” 格式。这个示例还表明，可以编写嵌套的函数调用。

接着，如果 `coalesce()` 函数的所有参数计算的值都是 `null`，则它会返回 `null`；否则它会返回第一个非空参数的值：

```
select i.name, coalesce(i.buyNowPrice, 0) from Item i

Root<Item> i = criteria.from(Item.class);
criteria.multiselect(
    i.get("name"),
    cb.coalesce(i.<BigDecimal>get("buyNowPrice"), 0)
);
```

如果一个 `Item` 不具有 `buyNowPrice`，则会为零值返回 `BigDecimal` 而非 `null`。

类似于 `coalesce()` 但更为强大的是 `case/when` 表达式。以下查询会返回每个 `User` 的 `username` 以及一个具有 “Germany” “Switzerland” 或 “Other” 的附加 `String`，这取决于该用户地址 `zipcode` 的长度：

```

select
    u.username,
    case when length(u.homeAddress.zipcode) = 5 then 'Germany'
         when length(u.homeAddress.zipcode) = 4 then 'Switzerland'
         else 'Other'
    end
from User u

// Check String literal support; see Hibernate bug HHH-8124
Root<User> u = criteria.from(User.class);
criteria.multiselect(
    u.get("username"),
    cb.selectCase()
        .when(
            cb.equal(
                cb.length(u.get("homeAddress").<String>get("zipcode")), 5
            ), "Germany"
        )
        .when(
            cb.equal(
                cb.length(u.get("homeAddress").<String>get("zipcode")), 4
            ), "Switzerland"
        )
        .otherwise("Other")
);

```

对于内置的标准函数，可以参阅上一节中的表。不同于限制中的函数调用，Hibernate 不会将投影中的一个未知函数调用传递给数据库作为一个普通直接的 SQL 函数调用。希望在投影中调用的任何函数都必须为 Hibernate 所知晓和/或用 JPQL 的特殊 `function()` 操作来调用。

此投影会返回每个拍卖 Item 的名称并且调用 H2 数据库的 SQL `datediff()` 函数返回商品创建和拍卖结束之间的天数：

```

select
    i.name,
    function('DATEDIFF', 'DAY', i.createdOn, i.auctionEnd)
from Item i

Root<Item> i = criteria.from(Item.class);
criteria.multiselect(
    i.get("name"),
    cb.function(
        "DATEDIFF",
        Integer.class,
        cb.literal("DAY"),
        i.get("createdOn"),
        i.get("auctionEnd")
    )
);

```


如果希望直接调用一个函数，则要为 Hibernate 提供该函数的返回类型，这样它就能解析该查询。要通过扩展配置的 `org.hibernate.Dialect` 来为投影中的调用添加函数。H2 方言中已经注册了 `datediff()` 函数。之后，可以像我们介绍的那样使用 `function()` 调用它，这在访问 H2 时适用于其他 JPA 提供程序，或者直接像 `datediff()` 那样调用，这很可能仅适用于 Hibernate。检查数据库方言的源代码；你可能会发现其中已经注册了许多其他专有 SQL 函数。

此外，可以在启动时通过调用 `HibernateMetadataBuilder` 上的 `applySqlFunction()` 方法来程式化地将 SQL 函数添加到 Hibernate。以下示例在 Hibernate 启动之前将 SQL 函数 `lpad()` 添加到了 Hibernate：

```
...
MetadataBuilder metadataBuilder = metadataSources.getMetadataBuilder();
metadataBuilder.applySqlFunction(
    "lpad",
    new org.hibernate.dialect.function.StandardSQLFunction(
        "lpad", org.hibernate.type.StringType.INSTANCE
    )
);
```

请查阅 `SQLFunction` 及其子类的 Javadoc 以了解更多信息。

接下来，我们要查看聚合函数，这是报告查询中最有用的函数。

15.3.5 聚合函数

报告查询利用了数据库的功能来执行数据的有效分组与聚合。例如，一个典型的报告查询会检索指定类别中最高的初始商品价格。此计算会发生在数据库中，并且不必将许多 `Item` 实体实例加载到内存中。

JPA 中标准化的聚合函数是 `count()`、`min()`、`max()`、`sum()` 和 `avg()`。

以下查询会计算所有 `Item` 的数量：

```
select count(i) from Item i

criteria.select(
    cb.count(criteria.from(Item.class))
);
```

该查询会将结果作为一个 `Long` 返回：

```
Long count = (Long)query.getSingleResult();
```

特殊的 `count(distinct)` JPQL 函数和 `countDistinct()` 方法会忽略重复项：

```
select count(distinct i.name) from Item i

criteria.select(
    cb.countDistinct(
        criteria.from(Item.class).get("name")
    )
);
```

```
);
```

以下查询会计算所有 Bid 的合计值：

```
select sum(b.amount) from Bid b
```

```
CriteriaQuery<Number> criteria = cb.createQuery(Number.class);
criteria.select(
    cb.sum(
        criteria.from(Bid.class).<BigDecimal>get("amount")
    )
);
```

此查询会返回一个 `BigDecimal`，因为 `amount` 属性是 `BigDecimal` 类型。`sum()` 函数还会识别 `BigInteger` 属性类型并且为所有其他数值属性类型返回 `Long`。

下面的查询会返回特定 Item 的最小和最大出价金额：

```
select min(b.amount), max(b.amount) from Bid b
       where b.item.id = :itemId
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.multiselect(
    cb.min(b.<BigDecimal>get("amount")),
    cb.max(b.<BigDecimal>get("amount"))
);
criteria.where(
    cb.equal(
        b.get("item").<Long>get("id"),
        cb.parameter(Long.class, "itemId")
    )
);
```

其结果是一个排序后的 `BigDecimal` 配对(一个 `Object[]` 数组中两个 `BigDecimal` 的实例)。

当在 `SELECT` 子句中调用一个聚合函数，而不是在 `GROUP BY` 子句中指定任何分组时，就会将结果大大降低为单个行，其中包含聚合后的值。这意味着(缺少 `GROUP BY` 子句时)任何包含一个聚合函数的 `SELECT` 子句都必须仅包含聚合函数。

对于更高级的统计和报告来说，需要能够执行分组，这是下一个主题。

15.3.6 分组

JPA 标准化了几个最常用于报告的 SQL 功能——尽管它们也可用于其他任务。在报告查询中，要编写用于投影的 `SELECT` 子句以及用于聚合的 `GROUP BY` 和 `HAVING` 子句。

就像在 SQL 中一样，任何出现在 `SELECT` 子句中的聚合函数之外的属性或别名也都必须出现在 `GROUP BY` 子句中。思考下面这个查询，它会统计各个姓的用户数量：

```
select u.lastname, count(u) from User u
       group by u.lastname
```

```
Root<User> u = criteria.from(User.class);
criteria.multiselect(
    u.get("lastname"),
    cb.count(u)
);
criteria.groupBy(u.get("lastname"));
```

在这个示例中，`u.lastname` 属性没有位于一个聚合函数内，因此所投影的数据必须是根据 `u.lastname` 分组的。也不需要指定你希望计数的属性；`count(u)` 表达式会被自动转译成 `count(u.id)`。

下面这个查询会为每个 Item 找出 `Bid#amount` 的平均值：

```
select i.name, avg(b.amount)
    from Bid b join b.item i
    group by i.name

Root<Bid> b = criteria.from(Bid.class);
criteria.multiselect(
    b.get("item").get("name"),
    cb.avg(b.<BigDecimal>get("amount"))
);
criteria.groupBy(b.get("item").get("name"));
```

Hibernate 特性

在分组时，你可能会遇到一个 **Hibernate** 限制。以下查询是符合规范的，但不能被 **Hibernate** 正确处理：

```
select i, avg(b.amount)
    from Bid b join b.item i
    group by i
```

JPA 规范允许根据实体路径表达式 `group by i` 来分组。但 **Hibernate** 不会在生成的 **SQL** **GROUP BY** 子句中自动扩展 `Item` 的属性，这样就无法匹配 **SELECT** 子句。必须在查询中手动扩展分组的/投影的属性，直到这个 **Hibernate** 问题被修复(这是一个最老并且持续时间最长的 **Hibernate** 问题，**HHH-1615**)：

```
select i, avg(b.amount)
    from Bid b join b.item i
    group by i.id, i.name, i.createdOn, i.auctionEnd,
           i.auctionType, i.approved, i.buyNowPrice,
           i.seller
```

```
Root<Bid> b = criteria.from(Bid.class);
Join<Bid, Item> i = b.join("item");
criteria.multiselect(
    i,
    cb.avg(b.<BigDecimal>get("amount"))
);
criteria.groupBy(
```



```
i.get("id"), i.get("name"), i.get("createdOn"), i.get("auctionEnd"),
i.get("auctionType"), i.get("approved"), i.get("buyNowPrice"),
i.get("seller")
);
```

有时希望通过只选择一个分组的特定值来进一步限制结果。使用 `WHERE` 子句在行上面执行限制的关系操作。`HAVING` 子句会在分组上执行限制。

例如，下面这个查询会计算每个以“D”开头的姓的用户数量：

```
select u.lastname, count(u) from User u
group by u.lastname
having u.lastname like 'D%'
```

```
Root<User> u = criteria.from(User.class);
criteria.multiselect(
    u.get("lastname"),
    cb.count(u)
);
criteria.groupBy(u.get("lastname"));
criteria.having(cb.like(u.<String>get("lastname"), "D%"));
```

`SELECT` 和 `HAVING` 子句遵循相同的规则：只有分组的属性可以出现在一个聚合函数之外。

前面几节应该让你理解了基础查询。是时候看一些更复杂的选项了。对于许多工程师来说，最难以理解却又是关系模型最大的好处在于，联结任意数据的能力。

15.4 联结

联结操作会用两种(或更多种)关系组合数据。在一个查询中联结数据可以在单个查询中抓取几个关联的实例与集合：例如，为了在与数据库的一次交互中加载一个 `Item` 及其所有的 `bids`。现在介绍基础联结操作如何工作以及如何使用它们来编写这样的动态抓取策略。首先来看看不使用 `JPA` 时，在 `SQL` 查询中联结是如何工作的。

15.4.1 使用 `SQL` 进行联结

我们首先处理已经介绍过的示例：联结 `ITEM` 和 `BID` 表中的数据，如图 15-1 所示。该数据库包含 3 个商品：第一个有三次出价，第二个有一次出价，而第三个没有出价。注意我们没有显示所有的列；因此有虚线。

大多数人在 `SQL` 数据库语境中听到联结一词时都会认为是一个内联结。内联结是几种联结类型中最重要的一种，并且最易于理解。思考图 15-2 中的 `SQL` 语句和结果。这个 `SQL` 语句的 `FROM` 子句中包含一个 `ANSI` 风格的内联结。

ITEM			BID			
ID	NAME	...	ID	ITEM_ID	AMOUNT	...
1	Foo	...	1	1	99.00	...
2	Bar	...	2	1	100.00	...
3	Baz	...	3	1	101.00	...
			4	2	4.99	...

图 15-1 ITEM 和 BID 表是联结操作显而易见的候选

```
select i.*, b.*
from ITEM i
inner join BID b on i.ID = b.ITEM_ID
```

i.ID	i.NAME	...	b.ID	b.ITEM_ID	b.AMOUNT
1	Foo	...	1	1	99.00
1	Foo	...	2	1	100.00
1	Foo	...	3	1	101.00
2	Bar	...	4	2	4.99

图 15-2 两个表的 ANSI 风格的内联结的结果

如果使用一个内联结来联结 ITEM 和 BID 表，并且使用 ITEM 行的 ID 必须匹配 BID 行的 ITEM_ID 值这一条件，那么就会在结果中得到商品与其出价的组合。注意此操作的结果只包含具有出价的商品。

可以认为一个联结会像下面这样工作：首先要使用两个表的积，得到 ITEM 行与 BID 行的所有可能组合。其次，要用一个联结条件过滤这些组合后的行：ON 子句中的表达式(任何好的数据库引擎都具有更复杂的算法来计算一个联结；它通常不会构建一个消耗内存的积，然后过滤出行)。联结条件是一个 Boolean 表达式，如果组合的行被包含在结果中，则会计算出结果为 true。

理解联结条件可以是计算出结果为 true 的任何表达式这一点至关重要。可以用任意方式联结数据；并不局限于标识符值的比较。例如，on i.ID = b.ITEM_ID and b.AMOUNT > 100 这个联结条件只会包含来自 BID 表同时还具有一个大于 100 的 AMOUNT 的行。BID 表中的 ITEM_ID 列具有一个外键约束，以确保 BID 具有对 ITEM 行的引用。这并不意味着你只能通过比较主键和外键列来进行联结。键列当然是联结条件中最常用的操作符，因为你通常希望将相关的信息一起检索。

如果想要所有的商品，而不仅仅是具有相关出价的商品，并且在没有对应出价时用 NULL 替代出价数据，那么就要编写一个(左)外联结，如图 15-3 所示。

```
select i.*, b.*
from ITEM i
left outer join BID b on i.ID = b.ITEM_ID
```

i.ID	i.NAME	...	b.ID	b.ITEM_ID	b.AMOUNT
1	Foo	...	1	1	99.00
1	Foo	...	2	1	100.00
1	Foo	...	3	1	101.00
2	Bar	...	4	2	4.99
3	Baz	...			

图 15-3 两个表的 ANSI 风格左外联结的结果

如果是左外联结，那么(左侧)ITEM 表中不满足联结条件的每一行也都会包含在结果中，为 BID 的所有列返回 NULL。右联结很少使用；开发人员总是从左到右进行思考，并且首先放置联结操作的“驱动”表。在图 15-4 中，可以看到使用 BID 替代 ITEM 作为驱动表的相同结果，以及一个右外联结。

select b.*, i.*
from BID b
right outer join ITEM i on b.ITEM_ID = i.ID

b.ID	b.ITEM_ID	b.AMOUNT	i.ID	i.NAME	...
1	1	99.00	1	Foo	...
2	1	100.00	1	Foo	...
3	1	101.00	1	Foo	...
4	2	4.99	2	Bar	...
			3	Baz	...

图 15-4 两个表的 ANSI 风格右外联结的结果

在 SQL 中，通常要显式指定联结条件。遗憾的是，使用外键约束的名称来指定如何联结两个表是不可行的：select * from ITEM join BID on FK_BID_ITEM_ID 无法生效。

要在 ON 子句中为一个 ANSI 风格的联结或者在 WHERE 子句中为一个所谓的 theta 风格联结指定联结条件：select * from ITEM i, BID b where i.ID = b.ITEM_ID。这是一个内联结；此处你会看到积是首先在 FROM 子句中创建的。

现在我们探讨 JPA 联结操作。记住，Hibernate 最终会将所有的查询转换成 SQL，因此即便语法稍微不同，但你也应该总是参考这一节中所示的图例并且验证你是否理解了所产生的 SQL 和结果集看起来会是什么样子。

15.4.2 JPA 中的联结选项

JPA 提供了查询中表述(内和外)联结的四种方式：

- 使用路径表达式的隐式关联联结
- 在 FROM 子句中使用 join 操作符的普通联结
- 在 FROM 子句中使用 join 操作符以及用于急抓取的 fetch 关键字的抓取联结
- WHERE 子句中的 theta 风格联结

我们首先介绍隐式关联联结。

15.4.3 隐式关联联结

在 JPA 查询中，不必显式指定一个联结条件。而是指定一个映射的 Java 类关联的名称。这是我们在 SQL 中使用的相同功能：用外键约束名称表述的联结条件。因为已经映射了数据库架构的大多数外键关系(如果不是所有的话)，所以可以在查询语言中使用这些映射关联的名称。这是一个语法糖，但它很方便。

例如，Bid 实体类具有一个名称为 item 的映射到 Item 实体类的多对一关联。如果在查询中选用这个关联，那么 Hibernate 就有足够的信息来减少使用键列比较的联结表述。这有

助于减少查询的谓词并且让其更具可读性。

在本章早前的内容中，我们介绍了使用圆点符号的属性路径表达式：像 `user.homeAddress.zipcode` 这样的单值路径表达式以及像 `item.bids` 这样的集合值路径表达式。可以在一个隐式内联结查询中创建一个路径表达式：

```
select b from Bid b where b.item.name like 'Fo%'
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.like(
        b.get("item").<String>get("name"),
        "Fo%"
    )
);
```

路径 `b.item.name` 会基于从 `Bid` 到 `Item` 的多对一关联创建一个隐式联结——此关联的名称是 `item`。Hibernate 清楚你使用 `BID` 表中的 `ITEM_ID` 外键映射了此关联并且对应生成了 SQL 联结条件。隐式联结总是会沿着多对一或者一对一关联的方向，不会遵循一个集合值关联(你不能编写 `item.bids.amount`)。

单路径表达式中可以使用多个联结：

```
select b from Bid b where b.item.seller.username = 'johndoe'
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.equal(
        b.get("item").get("seller").get("username"),
        "johndoe"
    )
);
```

此查询联结了来自 `BID`、`ITEM` 以及 `USER` 表的行。

我们不赞成将这个语法糖用于更复杂的查询。SQL 联结很重要，尤其是在优化查询时，需要能够一眼看到到底有多少联结。思考以下查询：

```
select b from Bid b where b.item.seller.username = 'johndoe'
and b.item.buyNowPrice is not null
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.and(
        cb.equal(
            b.get("item").get("seller").get("username"),
            "johndoe"
        ),
        cb.isNotNull(b.get("item").get("buyNowPrice"))
    )
);
```

SQL 中需要多少联结来表述这样一个查询呢？即便你正好得到了这个答案，但也需要更多一些时间来弄明白。答案是两个。所生成的 SQL 看起来会像这样：

```
select b.*
  from BID b
    inner join ITEM i on b.ITEM_ID = i.ID
    inner join USER u on i.SELLER_ID = u.ID
 where u.USERNAME = 'johndoe'
       and i.BUYNOWPRICE is not null;
```

或者，相较于联结这样复杂的路径表达式，可以在 FROM 子句中显式编写普通的联结。

15.4.4 显式联结

JPA 会在你可能用联结来达到的目的之间做出区别。假定你正在查询商品；你对于将它们与出价联结起来可能感兴趣的原因有两个。

你可能希望根据应用到其出价的一些条件来限制由查询返回的商品。例如，你可能想要具有出价大于 100 的所有商品，这就需要一个内联结。此处，你不关心不具有出价的商品。

另一方面，你可能主要关注的是商品，但可能希望执行一个外联结，原因只是你希望在单个 SQL 语句中为查询的商品检索所有出价，这就是我们之前所说的急联结抓取。记住，你希望默认延迟映射所有的关联，因此急抓取查询将在运行时为特定用例重写该默认抓取策略。

我们首先编写将联结用于限制目的的一些查询。如果希望检索 Item 实例并且将结果限制为具有某特定金额出价的商品，那么就必须将一个别名指定给联结的关联。然后你要在 WHERE 子句中引用该别名来限制你想要的数据：

```
select i from Item i
  join i.bids b
 where b.amount > 100
```

```
Root<Item> i = criteria.from(Item.class);
Join<Item, Bid> b = i.join("bids");
criteria.select(i).where(
    cb.gt(b.<BigDecimal>get("amount"), new BigDecimal(100))
);
```

此查询将别名 b 指定给了 bids 集合并且将所返回的 Item 实例限制为那些 Bid#amount 大于 100 的实例。

到目前为止，你只编写了内联结。外联结通常用于动态抓取，我们稍后将对其进行探讨。有些时候，你希望用外联结编写一个简单查询，而不应用一个动态抓取策略。例如，以下语句会查询并检索不具有出价的商品，以及具有最小出价金额的商品：

```
select i, b from Item i
  left join i.bids b on b.amount > 100
```

```
Root<Item> i = criteria.from(Item.class);
Join<Item, Bid> b = i.join("bids", JoinType.LEFT);
b.on(
    cb.gt(b.<BigDecimal>get("amount"), new BigDecimal(100))
);
criteria.multiselect(i, b);
```

此查询会在一个 `List<Object[]>` 中返回排序后的 `Item` 和 `Bid` 对。

这个查询中的第一个新内容就是条件查询中的 `LEFT` 关键字和 `JoinType.LEFT`。可以在 JPQL 中选择性地编写 `LEFT OUTER JOIN` 与 `RIGHT OUTER JOIN`，但我们通常选择缩写形式。

第二个变化是在 `ON` 关键字之后的额外联结条件。如果将 `b.amount > 100` 表达式放置在 `WHERE` 子句中，则会将结果限制为具有出价的 `Item` 实例。这并非你此处想要的：你希望检索商品和出价，甚至不具有出价的物品。如果一个商品具有出价，则出价金额必须大于 100。通过在 `FROM` 子句中添加一个额外的联结条件，可以限制 `Bid` 实例并且仍然检索所有的 `Item` 实例，无论它们是否具有出价。

这是该额外联结条件转换成 SQL 的方式：

```
... from ITEM i
left outer join BID b
on i.ID = b.ITEM_ID and (b.AMOUNT > 100)
```

该 SQL 查询将总是包含所映射关联的隐式联结条件，`i.ID = b.ITEM_ID`。只能将额外的表达式附加到该联结条件。JPA 和 Hibernate 不支持不具有映射实体关联或集合的任意外联结。

Hibernate 有一个专有的 `WITH` 关键字，它与 JPQL 中的 `ON` 关键字相同。可以在比较老的代码示例中看到它，因为 JPA 最近在标准化 `ON`。

可以使用右外联结编写一个返回相同数据的查询，切换驱动表就行：

```
select b, i from Bid b
right outer join b.item i
where b is null or b.amount > 100
```

```
Root<Bid> b = criteria.from(Bid.class);
Join<Bid, Item> i = b.join("item", JoinType.RIGHT);
criteria.multiselect(b, i).where(
    cb.or(
        cb.isNull(b),
        cb.gt(b.<BigDecimal>get("amount"), new BigDecimal(100))
    )
);
```

此右外联结查询比你可能认为的更为重要。要在尽可能的情况下避免映射一个持久化集合。如果没有使用一个一对多 `Item#bids` 集合，就需要一个右外联结来检索所有的 `Item` 及其 `Bid` 实例。你要从“另一”端来驱动该查询：多对一 `Bid#item`。

左外联结也在急动态抓取中扮演重要角色。

15.4.5 使用联结进行动态抓取

前几节中介绍的所有查询都有一个共同点：所返回的 `Item` 实例具有一个名称为 `bids` 的集合。如果被映射为 `FetchType.LAZY` (用于集合的默认设置)，那么此 `@OneToMany` 集合就不会被初始化，并且在访问它时就会触发一个额外的 SQL 语句。对于所有的单值关联也是如此，比如每个 `Item` 的 `@ManyToOne` 关联 `seller`。默认情况下，Hibernate 会生成一个代理以及延迟且仅按需加载所关联的 `User` 实例。

有哪些选项可以变更此行为呢？首先，可以修改映射元数据中的抓取计划并且将一个集合或单值关联声明为 `FetchType.EAGER`。然后 Hibernate 会执行必要的 SQL 以确保总是加载期望的实例网络。这也意味着单个 JPA 查询可能会产生几个 SQL 操作！例如，`select i from Item i` 这个简单查询会触发额外的 SQL 语句来加载每个 `Item` 的 `bids`、每个 `Item` 的 `seller` 等。

在第 12 章中，我们提出过应该在映射元数据中使用延迟全局抓取计划，其中不应该在关联和集合映射上使用 `FetchType.EAGER`。然后，对于应用程序中的特定用例，要动态重写该延迟抓取计划并且尽可能有效地编写一个抓取所需数据的查询。例如，你没有理由需要几个 SQL 语句来抓取所有的 `Item` 实例以及初始化其 `bids` 集合，也没有理由用几个 SQL 语句检索每个 `Item` 的 `seller`。可以在单个 SQL 语句中使用联结操作同时做这些事情。

在 JPQL 中使用 `FETCH` 关键字以及在条件查询 API 中使用 `fetch()` 方法急抓取关联数据是可行的：

```
select i from Item i
    left join fetch i.bids
```

```
Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
criteria.select(i);
```

在图 15-3 中你已经看到过这样做所产生的 SQL 查询以及结果集。

此查询会返回一个 `List<Item>`；每个 `Item` 实例都完全初始化了其 `bids` 集合。这不同于由上一节中的查询返回的排序后的配对！

当心——你可能不期望得到上述查询返回的重复结果：

```
List<Item> result = query.getResultList();
assertEquals(result.size(), 5);
```

结果中有 3 个商品、4
次出价、5 “行”

```
Set<Item> distinctResult = new LinkedHashSet<Item>(result);
assertEquals(distinctResult.size(), 3);
```

内存中进行“唯一处理”

始终只有三个商品

确保你理解为何这些重复项会出现在结果 `List` 中。验证结果集中 `Item` “行”的数量，如图 15-3 所示。Hibernate 会保留这些行用于列示元素；可能需要纠正行的计数以便在用户界面中更为容易地呈现一个报表。

可以通过一个 `LinkedHashSet` 传递该结果 `List` 来过滤掉重复的 `Item` 实例，`LinkedHashSet`

不允许重复元素但会保留元素的顺序。另外, Hibernate 可以用 `DISTINCT` 操作和 `distinct()` 条件方法移除重复元素:

```
select distinct i from Item i
left join fetch i.bids
```

```
Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
criteria.select(i).distinct(true);
```

要理解, 在这个例子中 `DISTINCT` 操作不会在数据库中执行。SQL 语句中不会存在 `DISTINCT` 关键字。从概念上讲, 无法在 SQL `ResultSet` 级别移除重复行。Hibernate 会在内存中执行去重复, 正如可以使用 `LinkedHashSet` 手动处理一样。

还可以用相同语法预抓取多对一或一对一关联:

```
select distinct i from Item i
left join fetch i.bids b
join fetch b.bidder
left join fetch i.seller
```

```
Root<Item> i = criteria.from(Item.class);
Fetch<Item, Bid> b = i.fetch("bids", JoinType.LEFT);
b.fetch("bidder");
i.fetch("seller", JoinType.LEFT);
criteria.select(i).distinct(true);
```

非空外键列。内联结或外联结没什么区别

此查询会返回一个 `List<Item>`, 并且每个 `Item` 都具有其初始化后的 `bids` 集合, 也会加载每个 `Item` 的 `seller`。最后, 会加载每个 `Bid` 实例的 `bidder`。可以通过联结 `ITEM`、`BID` 以及 `USERS` 表的行在一个 SQL 查询中达成此目的。

如果编写不具有 `LEFT` 的 `JOIN FETCH`, 就会得到使用内联结的急加载(如果使用 `INNER JOIN FETCH` 也是如此)。如果肯定有一个抓取的值, 那么急内联结抓取就合理了: 一个 `Item` 必须具有一个 `seller`, 并且一个 `Bid` 必须具有一个 `bidder`。

在一个查询中应该急加载多少关联是有限制的, 并且应该在一次数据库交互中抓取多少数据也是有限制的。思考以下查询, 它初始化了 `Item#bids` 和 `Item#images` 集合:

```
select distinct i from Item i
left join fetch i.bids
left join fetch i.images
```

```
Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
i.fetch("images", JoinType.LEFT);
criteria.select(i).distinct(true);
```

笛卡尔积: 不好

这是一个不好的查询, 因为它会创建 `bids` 和 `images` 的笛卡尔积, 有可能非常巨大的结果集。我们在 12.2.2 节中介绍过这个问题。

总的说来, 查询中的急动态抓取具有以下注意事项:

- 不要为进一步限制或投影将一个别名指定给任何抓取联结的关联或集合。left join fetch i.bids b where b.amount...这个查询是无效的。你不能认为，“加载 Item 实例并且初始化 bids 集合，但只具有包含特定金额的 Bid 实例。”可以将一个别名指定到一个抓取联结的关联以便进一步抓取：例如，检索每个 Bid 的 bidder: left join fetch i.bids b join fetch b.bidder。
- 不应该抓取多个集合；否则，就会创建笛卡尔积。可以随意抓取许多单值的关联，而不会创建积。
- 查询会忽略你在映射元数据中使用@org.hibernate.annotations.Fetch 定义的所有抓取策略。例如，使用 org.hibernate.annotations.FetchMode.JOIN 映射 bids 集合对于你编写的查询没有任何影响。你的查询的动态抓取策略会忽略全局抓取策略。另一方面，Hibernate 不会忽略映射的抓取计划：Hibernate 总是会考虑使用 FetchType.EAGER，并且你在执行查询时可能会看到几个额外的 SQL 语句。
- 如果急抓取一个集合，那么由 Hibernate 返回的 List 就会保留 SQL 结果中的行数作为重复引用。可以手动使用 LinkedHashSet 或者在查询中使用特殊的 DISTINCT 操作在内存中过滤掉重复项。

还有一个要注意的问题，并且它值得一些特殊关注。如果急抓取一个集合，则不能在数据库级别对结果集进行分页。例如，对于 select i from Item i fetch i.bids 这个查询，应该如何处理 Query#setFirstResult(21)和 Query#setMaxResults(10)呢？

显然，你只希望得到从商品 21 开始的 10 个商品。但也希望急加载每个 Item 的所有 bids。因此，数据库无法进行分页操作；不能将 SQL 结果限制为 10 个任意行。如果在一个查询中急抓取一个集合，那么 Hibernate 就会在内存中执行分页。这意味着所有的 Item 实例都会被加载到内存中，每一个都具有完全初始化的 bids 集合。然后 Hibernate 会为你提供所请求的商品页：例如，只包含商品 21 到 30。

并非所有商品都能放入内存中，并且你很可能期望在数据库中将结果传递给应用程序之前就在数据库中产生分页！因此，如果查询包含 fetch [collectionPath]并且你调用 setFirstResult()或 setMaxResults()，那么 Hibernate 会记录一条警告消息。

我们不建议使用带有 setMaxResults()或 setFirstResult()选项的 fetch [collectionPath]。通常可以编写一个更容易的查询来得到你希望呈现的数据——并且我们不期望你逐页加载数据来修改它。例如，如果希望显示几页商品并且为每个商品显示出价次数，则要编写一个报告查询：

```
select i.id, i.name, count(b)
  from Item i left join i.bids b
 group by i.id, i.name
```

可以使用 setFirstResult()和 setMaxResults()通过数据库对这个查询的结果进行分页。它比任何 Item 或 Bid 实例检索到内存中更为高效，因此就让数据库为你创建报告吧。

要介绍的最后一个 JPA 联结选项是 theta 风格的联结。

15.4.6 theta 风格的联结

在传统的 SQL 中，theta 风格的联结是 WHERE 子句中 with 联结条件一起的一个笛卡尔积，该子句被应用到积上以限制结果。在 JPA 查询中，在你的联结条件并不是映射到类关联的一个外键关系时，就可以使用 theta 风格语法。

例如，假定你在日志记录中存储 User 的姓名而不是映射从 LogRecord 到 User 的一种关联。这些类并不清楚彼此之间的任何事情，因为它们不是关联的。然后可以使用以下 theta 风格的联结找出所有的 User 及其 LogRecord：

```
select u, log from User u, LogRecord log
where u.username = log.username

Root<User> u = criteria.from(User.class);
Root<LogRecord> log = criteria.from(LogRecord.class);
criteria.where(
    cb.equal(u.get("username"), log.get("username")));
criteria.multiselect(u, log);
```

此处的联结条件是 username 的一个比较，作为一个属性出现在这两个类中。如果两个行都具有相同的 username，那么它们就会被联结(使用一个内联结)到结果中。该查询结果由排序过的配对构成：

```
List<Object[]> result = query.getResultList();
for (Object[] row : result) {
    assertTrue(row[0] instanceof User);
    assertTrue(row[1] instanceof LogRecord);
}
```

你很可能无须经常使用 theta 风格的联结。注意，目前 JPA 中还不能外联结两个不具有已映射关联的表——theta 风格的联结是内联结。

theta 风格的联结的另一个更常见用例是，在 WHERE 子句中将主键或外键值与查询参数或其他主键或外键值做比较：

```
select i, b from Item i, Bid b
where b.item = i and i.seller = b.bidder

Root<Item> i = criteria.from(Item.class);
Root<Bid> b = criteria.from(Bid.class);
criteria.where(
    cb.equal(b.get("item"), i),
    cb.equal(i.get("seller"), b.get("bidder"))
);
criteria.multiselect(i, b);
```

此查询会返回 Item 和 Bid 实例的配对，其中 bidder 也是 seller。这是 CaveatEmptor 中的一个重要查询，因为它可以检测出对自己商品出价的人。你很可能应该将这个查询转换成一个数据库约束并且不允许存储这样的 Bid 实例。

上面的查询还有一个有意思的比较表达式: `i.seller = b.bidder`。这是一个标识符比较, 是我们的下一个主题。

15.4.7 比较标识符

JPA 支持在查询中使用以下隐式标识符比较语法:

```
select i, u from Item i, User u
       where i.seller = u and u.username like 'j%'
```

```
Root<Item> i = criteria.from(Item.class);
Root<User> u = criteria.from(User.class);
criteria.where(
    cb.equal(i.get("seller"), u),
    cb.like(u.<String>get("username"), "j%")
);
criteria.multiselect(i, u);
```

在此查询中, `i.seller` 指的是 `ITEM` 表的 `SELLER_ID` 外键列, 引用了 `USERS` 表。别名 `u` 指的是 `USERS` 表的主键(`ID` 列)。因此, 此查询具有一个 `theta` 风格的联结并且等同于更容易的、可阅读的替代项:

```
select i, u from Item i, User u
       where i.seller.id = u.id and u.username like 'j%'
```

```
Root<Item> i = criteria.from(Item.class);
Root<User> u = criteria.from(User.class);
criteria.where(
    cb.equal(i.get("seller").get("id"), u.get("id")),
    cb.like(u.<String>get("username"), "j%")
);
criteria.multiselect(i, u);
```

Hibernate 特性

以 `id` 结尾的路径表达式在 `Hibernate` 中是特殊的: `id` 这个名称总是指向一个实体的标识符属性。用 `@Id` 注解的属性的实际名称是什么并不重要; 总是可以使用 `entityAlias.id` 得到它。这就是我们建议你总是将你的实体类的标识符属性命名为 `id` 的原因, 以避免在查询中混淆。注意, 这在 `JPA` 中并非是必要的要求或者标准; 只有 `Hibernate` 会将一个 `id` 路径元素作特殊处理。

你还可能希望将一个键值与一个查询参数作比较, 也许是为了找出指定 `seller` (一个 `User`) 的所有 `Item`:

```
select i from Item i where i.seller = :seller
```

```
Root<Item> i = criteria.from(Item.class);
criteria.where(
    cb.equal(
```

```

        i.get("seller"),
        cb.parameter(User.class, "seller")
    )
);
criteria.select(i);
query.setParameter("seller", someUser);
List<Item> result = query.getResultList();

```

另外，根据标识符值而不是对象引用来表述这些种类的查询。这些查询等同于之前的查询：

```

select i from Item i where i.seller.id = :sellerId

Root<Item> i = criteria.from(Item.class);
criteria.where(
    cb.equal(
        i.get("seller").get("id"),
        cb.parameter(Long.class, "sellerId")
    )
);
criteria.select(i);
query.setParameter("sellerId", USER_ID);
List<Item> result = query.getResultList();

```

思考一下标识符属性，下面这个查询对与其后看起来类似的查询对之间存在天壤之别：

```

select b from Bid b where b.item.name like 'Fo%'

Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.like(
        b.get("item").<String>get("name"),
        "Fo%"
    )
);

```

和这个看起来类似的查询对：

```

select b from Bid b where b.item.id = :itemId

CriteriaQuery<Bid> criteria = cb.createQuery(Bid.class);
Root<Bid> b = criteria.from(Bid.class);
criteria.where(
    cb.equal(
        b.get("item").get("id"),
        cb.parameter(Long.class, "itemId")
    )
);
criteria.select(b);

```

第一个查询对使用了隐式表联结；第二个完全没有使用联结！

到此我们就完成了对于涉及联结的查询的探讨。我们的最后一个主题是查询中的嵌套查询：子查询。

15.5 子查询

子查询是 SQL 的一个重要且强大的功能。子查询是嵌入到另一个查询中的选择查询，通常是嵌入到 SELECT、FROM 或 WHERE 子句中。

JPA 支持 WHERE 子句中的子查询。FROM 子句中的子查询不受支持，因为查询语言没有传递闭包。一个查询的结果不可用于 FROM 子句中的进一步查询。查询语言也不支持 SELECT 子句中的子查询，但可以使用 `@org.hibernate.annotations.Formula` 将子查询映射到派生属性，如 5.1.3 节中所介绍的那样。

子查询可以与查询的其余部分相关，也可以不相关。

15.5.1 相关与不相关的嵌套

一个子查询的结果可能包含单行或者多行。通常，返回单行的子查询会执行聚合。以下子查询会返回一个用户售出的商品总数；该外查询返回已经售出多个商品的用户：

```
select u from User u
where (select count(i) from Item i where i.seller = u
) > 1
```

```
Root<User> u = criteria.from(User.class);
```

```
Subquery<Long> sq = criteria.subquery(Long.class);
```

```
Root<Item> i = sq.from(Item.class);
```

```
sq.select(cb.count(i))
    .where(cb.equal(i.get("seller"), u)
);
```

```
criteria.select(u);
```

```
criteria.where(cb.greaterThan(sq, 1L));
```

此内查询是一个相关的子查询——它指的是一个来自外查询的别名(u)。

下面这个查询包含一个不相关的子查询：

```
select b from Bid b
where b.amount + 1 >= (
    select max(b2.amount) from Bid b2
)
```

```
Root<Bid> b = criteria.from(Bid.class);
```

```
Subquery<BigDecimal> sq = criteria.subquery(BigDecimal.class);
```

```
Root<Bid> b2 = sq.from(Bid.class);
```

```
sq.select(cb.max(b2.<BigDecimal>get("amount")));
```

```
criteria.select(b);
criteria.where(
    cb.greaterThanOrEqualTo(
        cb.sum(b.<BigDecimal>get("amount"), new BigDecimal(1)),
        sq
    )
);
```

这个示例中的子查询会返回整个系统中的最大出价金额；此外查询会返回那些金额在该金额的一个单位(美元、欧元等)以内的所有出价。注意在这两种情况下，JPQL 中的子查询都位于圆括号中。这总是必须的。

不相关的子查询没什么危害，并且在方便时没有理由不使用它们。总是可以将它们重写成两个查询，因为它们不会彼此引用。应该更为仔细地考虑与相关子查询的性能影响有关的事情。在一个成熟的数据库上，一个简单相关子查询的性能开销类似于一个联结的开销。但不可能使用几个独立查询来重写一个相关子查询。

如果一个子查询返回多行，那么就可以用量化来合并它。

15.5.2 量化

以下量词限定符是标准的：

- **ALL**——如果对于子查询结果中所有值的比较为真，则表达式计算结果为 **true**。如果子查询结果有一个值未通过比较验证，则它的计算结果就为 **false**。
- **ANY**——如果对于子查询结果中的一些(任何)值的比较为真，则表达式计算结果为 **true**。如果子查询结果为空或没有值满足该比较，则它的计算结果就为 **false**。关键字 **SOME** 是 **ANY** 的同义词。
- **EXISTS**——如果子查询的结果由一个或多个值构成，则计算结果为 **true**。

例如，以下查询会返回所有出价都小于或等于 10 的商品：

```
select i from Item i
where 10 >= all (
    select b.amount from i.bids b
)
```

```
Root<Item> i = criteria.from(Item.class);
Subquery<BigDecimal> sq = criteria.subquery(BigDecimal.class);
Root<Bid> b = sq.from(Bid.class);
sq.select(b.<BigDecimal>get("amount"));
sq.where(cb.equal(b.get("item"), i));

criteria.select(i);
criteria.where(
    cb.greaterThanOrEqualTo(
        cb.literal(new BigDecimal(10)),
        cb.all(sq)
    )
);
```

以下查询会返回有一次出价正好为 101 的商品：

```
select i from Item i
  where 101.00 = any (
    select b.amount from i.bids b
  )
```

```
Root<Item> i = criteria.from(Item.class);
Subquery<BigDecimal> sq = criteria.subquery(BigDecimal.class);
Root<Bid> b = sq.from(Bid.class);
sq.select(b.<BigDecimal>get("amount"));
sq.where(cb.equal(b.get("item"), i));

criteria.select(i);
criteria.where(
  cb.equal(
    cb.literal(new BigDecimal("101.00")),
    cb.any(sq)
  )
);
```

要检索所有具有出价的商品，可以检查使用 EXISTS 的子查询的结果：

```
select i from Item i
  where exists (
    select b from Bid b where b.item = i
  )
```

```
Root<Item> i = criteria.from(Item.class);
Subquery<Bid> sq = criteria.subquery(Bid.class);
Root<Bid> b = sq.from(Bid.class);
sq.select(b).where(cb.equal(b.get("item"), i));

criteria.select(i);
criteria.where(cb.exists(sq));
```

这个查询远比它看起来重要。可以使用后面这个查询找出所有具有出价的商品：select i from Item i where i.bids is not empty。不过，这需要一个映射的一对多集合 Item#bids。如果遵循我们的建议，则可能只映射了“另一”端：多对一 Bid#item。使用一个 exists() 和一个子查询，你会得到相同结果。

子查询是一项高级技术；应该时常考虑一下子查询的使用，因为具有子查询的查询通常只能用联结和聚合来重写。但它们常常是强大且有用的。

15.6 本章小结

- 如果你在阅读本章之前就了解了 SQL，那么现在就可以在 JPQL 中以及使用条件查询 API 来编写各种各样的查询。

高级查询选项

第 16 章

16

本章内容简介:

- 转换查询结果
- 过滤集合
- 使用 Hibernate API 按条件查询

本章会阐释可选或高级的查询选项: 转换查询结果、过滤集合以及 Hibernate 条件查询 API。首先探讨 Hibernate 的 ResultTransformer API, 使用它可以将一个结果转换器应用到查询结果以便使用自己的代码替代 Hibernate 的默认行为来过滤或封送处理结果。

在前面几章中, 我们总是建议你在映射集合时要注意, 因为它很少值得付出精力。在本章中, 我们会介绍集合过滤器, 一个让持久化集合更有价值的原生 Hibernate 功能。最后, 我们要查看另一个专有的 Hibernate 功能, org.hibernate.Criteria 查询 API, 以及你可能选用它来替代标准 JPA 按条件查询时的一些情境。

我们首先介绍查询结果的转换。

Hibernate 特性

16.1 转换查询结果

可以将一个结果转换器应用到查询结果, 以便可以使用你自己的程序替代 Hibernate 的默认行为来对结果进行过滤或封送处理。Hibernate 的默认行为提供了一组默认转换器, 可以替换它们和/或自定义。

你打算转换的结果是一个简单查询的结果, 但需要通过 Session 访问原生的 Hibernate API org.hibernate.Query, 如代码清单 16.1 所示。

代码清单 16.1 具有几个投影的元素的简单查询

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
Session session = em.unwrap(Session.class);
org.hibernate.Query query = session.createQuery(
    "select i.id as itemId, i.name as name, i.auctionEnd as auctionEnd from
```

```
Item i"
);
```

未使用任何自定义结果转换，这个查询就会返回 `Object[]` 的一个 `List`：

路径：/examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
List<Object[]> result = query.list();

for (Object[] tuple : result) {
    Long itemId = (Long) tuple[0];
    String name = (String) tuple[1];
    Date auctionEnd = (Date) tuple[2];
    // ...
}
```

每个对象数组都是该查询结果的一“行”。该元组的每个元素都可以通过索引来访问：此处索引 0 是一个 `Long`、索引 1 是一个 `String`、索引 2 是一个 `Date`。我们要介绍的第一个结果转换器会返回由多个 `List` 组成的一个列表。

转换条件查询结果

这一节中的所有示例都是用 `org.hibernate.Query` API 创建的在 JPQL 中编写的查询。如果使用一个 `CriteriaBuilder` 编写一个 `JPA CriteriaQuery`，则无法应用 `Hibernate org.hibernate.transform.ResultTransformer`：这是仅供 `Hibernate` 使用的接口。即使你得到了用于条件查询的原生 API（通过 `HibernateQuery` 转换，如 14.1.3 节中所示），你也无法设置一个自定义转换器。对于 `JPA CriteriaQuery`，`Hibernate` 会应用一个内置转换器来实现 `JPA` 契约；使用一个自定义转换器会重写该内置转换器并且出现问题。不过，可以在用 `HibernateQuery` 得到原生 API 之后，为使用 `javax.persistence.Query` 创建的 JPQL 查询设置一个自定义转换器。此外，在本章稍后的内容中，你会看到原生的 `org.hibernate.Criteria` API，一个可选的按条件查询实用工具，它支持重写 `org.hibernate.transform.ResultTransformer`。

16.1.1 返回一系列列表

我们假设你希望使用索引访问却又不想要 `Object[]` 结果。相较于 `Object[]` 的列表，每个元组还可以用一个 `List` 来表示，这要使用 `ToListResultTransformer`：

路径：/examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
query.setResultTransformer(
    ToListResultTransformer.INSTANCE
);

List<List> result = query.list();

for (List list : result) {
    Long itemId = (Long) list.get(0);
    String name = (String) list.get(1);
    Date auctionEnd = (Date) list.get(2);
    // ...
}
```


这是一个很小的区别，但如果应用程序其他层中的代码已经使用了一系列列表的话，那么它就是一个便利的替代项。

接下来的转换器会将查询结果转换为每个元组一个 `Map`，其中查询投影会指定映射到投影元素的别名。

16.1.2 返回一系列映射

`AliasToEntityMapResultTransformer` 会返回 `java.util.Map` 的一个 `List`，每“行”一个映射。该查询中的别名是 `itemId`、`name` 和 `auctionEnd`：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java`

```
query.setResultTransformer(
    AliasToEntityMapResultTransformer.INSTANCE
);

List<Map> result = query.list();

assertEquals(
    query.getReturnAliases(),
    new String[]{"itemId", "name", "auctionEnd"}
);

for (Map map : result) {
    Long itemId = (Long) map.get("itemId");
    String name = (String) map.get("name");
    Date auctionEnd = (Date) map.get("auctionEnd");
    // ...
}
```

← 访问查询别名

如果不知道查询中使用的别名并且需要动态获得它们，则可以调用 `org.hibernate.Query#getReturnAliases()`。

该示例查询会返回标量值；你也可能希望转换包含持久化实体实例的结果。这个示例为投影的实体和多个 `Map` 的一个 `List` 使用了别名：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java`

```
org.hibernate.Query entityQuery = session.createQuery(
    "select i as item, u as seller from Item i join i.seller u"
);

entityQuery.setResultTransformer(
    AliasToEntityMapResultTransformer.INSTANCE
);

List<Map> result = entityQuery.list();

for (Map map : result) {
    Item item = (Item) map.get("item");
    User seller = (User) map.get("seller");
    assertEquals(item.getSeller(), seller);
}
```

```
// ...  
}
```

接下来的转换器更有用，它会根据别名将查询结果映射到 JavaBean 属性。

16.1.3 将别名映射到 bean 属性

在 15.3.2 节中，我们介绍了一个查询如何才能通过调用 `ItemSummary` 构造函数动态返回一个 JavaBean 的实例。在 JPQL 中，要使用 `new` 操作符达成此目的。对于条件查询，要使用 `construct()` 方法。`ItemSummary` 类必须具有一个匹配已投影查询结果的构造函数。

或者，如果 JavaBean 不具有正确的构造函数，那么仍旧可以进行实例化并且使用 `AliasToBeanResultTransformer` 通过设置方法和/或字段填充其值。以下示例转换了代码清单 16.1 中的查询结果：

路径：/examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
query.setResultTransformer(  
    new AliasToBeanResultTransformer(ItemSummary.class)  
);  
  
List<ItemSummary> result = query.list();  
  
for (ItemSummary itemSummary : result) {  
    Long itemId = itemSummary.getItemId();  
    String name = itemSummary.getName();  
    Date auctionEnd = itemSummary.getAuctionEnd();  
    // ...  
}
```

用希望实例化的 JavaBean 类创建了该转换器，此处是 `ItemSummary`。Hibernate 要求这个类要么没有构造函数，要么有一个公共的无参构造函数。

在转换该查询结果时，Hibernate 会使用与结果中别名相同的名称来查找设置方法和字段。`ItemSummary` 类必须具有字段 `itemId`、`name` 和 `auctionEnd`，或者具有设置方法 `setItemId()`、`setName()` 和 `setAuctionEnd()`。字段或设置方法的参数必须是正确的类型。如果有映射到一些查询别名的字段以及映射到其余内容的设置方法，那也是可以的。

你还应该知道如何在内置转换器不满足你的需求时编写你自己的 `ResultTransformer`。

16.1.4 编写一个 ResultTransformer

Hibernate 中内置的转换器并不复杂；用列表、映射或对象数组表示的结果元组之间没有太大区别。不过，实现 `ResultTransformer` 接口只是很简单的事情，并且查询结果的自定义转换可以加强应用程序中的代码层之间的集成。如果用户界面代码已经知道如何呈现 `List<ItemSummary>` 的一个表，那么就可以让 Hibernate 直接从一个查询中返回它。

接下来，我们要介绍如何实现一个 `ResultTransformer`。我们假设你想要一个从代码清单 16.1 的查询中返回的 `List<ItemSummary>`，但你不能让 Hibernate 通过一个构造函数上的

反射来创建 `ItemSummary` 的实例。可能你的 `ItemSummary` 类被预定义过并且没有正确的构造函数、字段和设置方法。相反，你有一个 `ItemSummaryFactory` 来生成 `ItemSummary` 的实例。

`ResultTransformer` 接口要求你实现方法 `transformTuple()` 和 `transformList()`：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java`

```
query.setResultTransformer(
    new ResultTransformer() {
        ❶ 转换结果行
        @Override
        public Object transformTuple(Object[] tuple, String[] aliases) {
            Long itemId = (Long) tuple[0];
            String name = (String) tuple[1];
            Date auctionEnd = (Date) tuple[2];

            assertEquals(aliases[0], "itemId");
            assertEquals(aliases[1], "name");
            assertEquals(aliases[2], "auctionEnd");
            // 如果需要，就访问查询别名

            return ItemSummaryFactory.newItemSummary(
                itemId, name, auctionEnd
            );
        }
        ❷ 修改结果列表
        @Override
        public List transformList(List collection) {
            return Collections.unmodifiableList(collection);
        }
    }
);
```

Collection 是一个 `List<ItemSummary>`

❶ 对于每一个结果“行”，`Object[]`元组都必须被转换成该行所期望的结果值。此处通过元组中的索引访问每一个投影元素，然后调用 `ItemSummaryFactory` 生成查询结果值。`Hibernate` 会为每一个元组元素向该方法传递查询中找到的别名。不过，这个转换器中不需要别名。

❷ 可以在转换元组之后包装或修改结果列表。此处返回的 `List` 变成了不可修改：这对于无论如何都不应修改数据的报告界面来说是合适的。

正如可以在该示例中看到的，分两步转换了查询结果：首先自定义如何将查询结果的每一“行”或元组转换成你期望的任何值。然后处理了这些值的整个 `List`，再次包装或转换。

接下来，我们要探讨另一个便利的 `Hibernate` 功能(JPA 没有它的对等项)：集合过滤器。

Hibernate 特性

16.2 过滤集合

在第 7 章中介绍了应该(或者不应该)在 Java 域模型中映射一个集合的原因。集合映射

的主要好处是更易于访问数据：可以调用 `item.getImages()` 或 `item.getBids()` 来访问与一个 `Item` 相关的所有图片和出价。不必编写 JPQL 或条件查询；在开始遍历集合元素时，Hibernate 会执行查询。

使用此自动数据访问的最明显的问题是，Hibernate 总是会编写相同的查询，以便为一个 `Item` 检索所有的图片或出价。可以自定义集合元素的顺序，但即使这样，它也是一个静态映射。要根据创建日期升序和降序呈现一个 `Item` 的两组出价，你会怎么做？可以回到以前的方式编写和执行自定义查询并且不调用 `item.getBids()`；甚至可能不需要集合映射。

反之，可以使用一个 Hibernate 专有功能，集合过滤器，它会让这些查询的编写变得更容易，这要用到映射的集合。我们假设内存中有一个持久化 `Item` 实例，可能是用 `EntityManager` API 加载的。列出这个 `Item` 的所有 bids，但要进一步将结果限制为某个特定 `User` 做出的 bids。根据 `Bid#amount` 降序对列表进行排序。见代码清单 16.2。

代码清单 16.2 过滤和排序一个集合

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java`

```
Item item = em.find(Item.class, ITEM_ID);
User user = em.find(User.class, USER_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "where this.bidder = :bidder order by this.amount desc"
);

query.setParameter("bidder", user);
List<Bid> bids = query.list();
```

`session.createFilter()` 方法接受一个持久化集合和一个 JPQL 查询片段。这个查询片段不需要 `select` 或 `from` 子句；此处它只有一个使用 `where` 子句和 `order by` 子句的限制。别名 `this` 总是指向集合的元素，这里是 `Bid` 实例。所创建的过滤器是一个普通的 `org.hibernate.Query`，用一个绑定参数来准备好并且用 `list()` 来执行，就像往常一样。

Hibernate 不会在内存中执行集合过滤器。当调用该过滤器时，`Item#bids` 集合可能是未初始化的，并且如果是这样的话，它仍旧会保持未初始化状态。此外，过滤器不会应用到瞬时集合或查询结果。仅可以将它们应用到当前被一个由持久化上下文托管的实体实例所引用的已映射持久化集合。过滤器这个词会让人有点误解，因为过滤的结果是完全新的并且不同的集合；不会触及原始集合。

包括此功能的设计者在内，出乎每个人意料之外的是，即便是不重要的过滤器最后被证明都是有用的。例如，可以使用一个空查询来对集合元素分页：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java`

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    ""
);
```

```
query.setFirstResult(0);
query.setMaxResults(2);           ← 只检索两个 bids
List<Bid> bids = query.list();
```

这里，Hibernate 执行了该查询，加载集合元素并将返回行限制为从结果的零行开始的两行。通常，会用分页的查询使用一个 `order by`。

集合过滤器中不需要 `from` 子句，但可以使用一个 `from` 子句，如果这是你的习惯的话。集合过滤器甚至不需要返回被过滤的集合元素。

下面这个过滤器会返回由任何竞价者售出的任何 Item:

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java`

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "from Item i where i.seller = this.bidder"
);

List<Item> items = query.list();
```

使用一个 `select` 子句，就可以声明一个投影。以下过滤器会检索做了出价的用户姓名:

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java`

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "select distinct this.bidder.username order by this.bidder.username asc"
);

List<String> bidders = query.list();
```

这一切都很有意思，但集合过滤器存在的最重要原因是，让应用程序可以在不初始化整个集合的情况下检索集合元素。对于大型集合来说，实现可接受的性能是很重要的。以下查询会检索 Item 具有的金额大于等于 100 的所有 bids:

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java`

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "where this.amount >= :param"
);

query.setParameter("param", new BigDecimal(100));
List<Bid> bids = query.list();
```

同样，这不会初始化 `Item#bids` 集合，而是返回一个新的集合。

在 JPA 2 之前，按条件查询仅可用作一个专有的 Hibernate API。如今，标准化的 JPA 接口与老的 `org.hibernate.Criteria` API 同样强大，因此你很少会需要它。但有几个功能仍旧仅在 Hibernate API 中可用，比如按示例查询以及嵌入任意 SQL 片段。在下一节中，你会看到 `org.hibernate.Criteria` API 及其一些独特选项的简要概述。

Hibernate 特性

16.3 Hibernate 条件查询 API

使用 `org.hibernate.Criteria` 和 `org.hibernate.Example` 接口，就可以通过创建与合并 `org.hibernate.criterion.*` 实例来编程式构建查询。你会看到如何使用这些 API 以及如何表述选择、限制、联结和投影。我们假设你已经阅读过上一章并且你知道这些操作如何被转换成 SQL。这里显示的所有查询示例在上一章中都有一个对等的 JPQL 或 JPA 条件示例，因此如果需要比较所有三个 API 的话，就可以轻易地翻阅这两章。

我们首先介绍一些基础选择示例。

16.3.1 选择和排序

以下查询会加载所有的 Item 实例：

路径：`/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
org.hibernate.Criteria criteria = session.createCriteria(Item.class);
List<Item> items = criteria.list();
```

使用 Session 创建了一个 `org.hibernate.Criteria`。或者，可以不使用打开的持久化上下文创建一个 `DetachedCriteria`：

路径：`/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
DetachedCriteria criteria = DetachedCriteria.forClass(Item.class);
List<Item> items = criteria.getExecutableCriteria(session).list();
```

当准备好执行该查询时，要使用 `getExecutableCriteria()` 将其“附加”到一个 Session。注意这是 Hibernate 条件 API 的一个独特功能。使用 JPA，你总是至少需要一个 `EntityManagerFactory` 来得到一个 `CriteriaBuilder`。

可以声明结果的顺序，等同于 JPQL 中的 `order by` 子句。以下查询会加载按照名和姓升序排列的所有 User 实例：

路径：`/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
List<User> users =
    session.createCriteria(User.class)
        .addOrder(Order.asc("firstname"))
        .addOrder(Order.asc("lastname"))
```



```
.list();
```

在这个示例中，代码是以流畅风格编写的(使用方法链接)；像 `addOrder()` 这样的方法会返回原始的 `org.hibernate.Criteria`。

接下来，我们看看限制所选择的记录。

16.3.2 限制

以下查询会返回所有 `name` 为 “Foo” 的 `Item` 实例：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
List<Item> items =
    session.createCriteria(Item.class)
        .add(Restrictions.eq("name", "Foo"))
        .list();
```

`Restrictions` 接口是用于可以添加到 `Criteria` 的个体 `Criterion` 的工厂。属性是使用简单字符串来处理的，这里是使用 “name” 来处理 `Item#name` 的。

还可以匹配子字符串，类似于 JPQL 中的 `like` 操作符。以下查询会加载具有以 “j” 或 “J” 开头的 `username` 的所有 `User` 实例：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
List<User> users =
    session.createCriteria(User.class)
        .add(Restrictions.like("username", "j",
            MatchMode.START).ignoreCase())
        .list();
```

`MatchMode.START` 等同于 JPQL 中的通配符 `j%`。其他的模式是 `EXACT`、`END` 和 `ANYWHERE`。

可以使用圆点符号命名可嵌入类型的嵌套属性，比如 `User` 的 `Address`。

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
List<User> users =
    session.createCriteria(User.class)
        .add(Restrictions.eq("homeAddress.city", "Some City"))
        .list();
```

Hibernate `Criteria` API 的一个独特功能是能够在限制中编写普通 SQL 片段。这个查询会加载 `username` 少于 8 个字符的所有 `User` 实例：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
List<User> users =
    session.createCriteria(User.class)
        .add(Restrictions.sqlRestriction(
            "length({alias}.USERNAME) < ?",
```

```

8,
StandardBasicTypes.INTEGER
)).list();

```

Hibernate 会将该 SQL 片段原样发送给数据库。需要 {alias} 占位符在最终 SQL 中作为所用表的别名前缀；它总是指向根实体被映射到的表(这个例子中是 USERS)。还应用了一个位置参数(不被此 API 支持的命名参数)并且将其类型指定为 StandardBasicTypes.INTEGER。

扩展 Hibernate 的条件系统

Hibernate 条件查询系统是可扩展的：可以在你自己的 org.hibernate.criterion.Criterion 接口实现中包装 LENGTH() 这个 SQL 函数。

在执行选择和限制之后，希望将投影添加到你的查询，以声明你希望检索的数据。

16.3.3 投影和聚合

以下查询会返回所有 User 实体的标识符值、username 以及 homeAddress：

路径：/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```

List<Object[]> result =
    session.createCriteria(User.class)
        .setProjection(Projections.projectionList()
            .add(Projections.property("id"))
            .add(Projections.property("username"))
            .add(Projections.property("homeAddress"))
        ).list();

```

此查询的结果是 Object[] 的一个 List，每个元组一个数组。每个数组都包含一个 Long(或者无论该用户的标识符类型是什么)、一个 String 以及一个 Address。

就像使用限制一样，可以将任意 SQL 表达式和函数调用添加到投影：

路径：/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```

List<String> result =
    session.createCriteria(Item.class)
        .setProjection(Projections.projectionList()
            .add(Projections.sqlProjection(
                "NAME || ':' || AUCTIONEND as RESULT",
                new String[]{"RESULT"},
                new Type[]{StandardBasicTypes.STRING}
            ))
        ).list();

```

此查询会返回 String 的一个 List，其中字符串具有 “[商品名称]:[拍卖结束日期]” 这一格式。该投影的第二个参数是你在查询中使用的别名的名称：Hibernate 需要这个来读取 ResultSet 的值。还需要每个投影的元素/别名的类型：这里是 StandardBasicTypes.STRING。

Hibernate 支持分组与聚合。此查询会计算用户的姓的数量：

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Object[]> result =
    session.createCriteria(User.class)
        .setProjection(Projections.projectionList()
            .add(Projections.groupProperty("lastname"))
            .add(Projections.rowCount())
        ).list();
```

rowCount()方法等同于 JPQL 中的 count()函数调用。以下聚合查询会返回按 Item 分组的 Bid 平均金额：

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Object[]> result =
    session.createCriteria(Bid.class)
        .setProjection(Projections.projectionList()
            .add(Projections.groupProperty("item"))
            .add(Projections.avg("amount"))
        ).list();
```

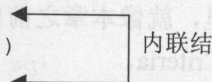
接下来，你会看到还可以在 Criteria API 中使用的联结。

16.3.4 联结

你用嵌套的 Criteria 表述了一个关联实体的内联结：

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Bid> result =
    session.createCriteria(Bid.class)
        .createCriteria("item")
        .add(Restrictions.isNotNull("buyNowPrice"))
        .createCriteria("seller")
        .add(Restrictions.eq("username", "johndoe"))
        .list();
```



此查询会返回所有由 User “johndoe” 售出的 Item 的 Bid 实例，User “johndoe” 没有 buyNowPrice。Bid#item 关联的第一个内联结是与 Bid 的根 Criteria 上的 createCriteria("item") 一起进行的。现在这个嵌套的 Criteria 表示关联路径，其上进行了与 createCriteria("seller") 的另一个内联结。每个联结 Criteria 上还放置了进一步的限制；会在最终 SQL 查询的 where 子句中使用逻辑 and 来合并它们。

另外，内联结可以用一个 Criteria 上的 createAlias()来表述。这是相同的查询：

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Bid> result =
    session.createCriteria(Bid.class)
        .createCriteria("item")
```

内联结


```

        .createAlias("seller", "s")
        .add(Restrictions.and(
            Restrictions.eq("s.username", "johndoe"),
            Restrictions.isNotNull("buyNowPrice")
        ))
        .list();

```

使用外联结的动态急抓取是使用 `setFetchMode()` 来声明的:

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```

List<Item> result =
    session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .list();

```

此查询会返回具有其在相同 SQL 查询中初始化的 `bids` 集合的所有 `Item` 实例。

当心重复

就像使用 JPQL 和 JPA 条件查询一样, Hibernate 可能会返回重复的 `Item` 引用! 参阅 15.4.5 节。

类似于 JPQL 和 JPA 条件, Hibernate 可以使用一个“唯一”操作过滤掉内存中的重复引用:

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```

List<Item> result =
    session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
        .list();

```

这里, 就像本章之前内容所探讨过的那样, 还可以看到 `ResultTransformer` 可以被应用到一个 `Criteria`。

可以在一个查询中抓取多个关联/集合:

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```

List<Item> result =
    session.createCriteria(Item.class)
        .createAlias("bids", "b", JoinType.LEFT_OUTER_JOIN)
        .setFetchMode("b", FetchMode.JOIN)
        .createAlias("b.bidder", "bdr", JoinType.INNER_JOIN)
        .setFetchMode("bdr", FetchMode.JOIN)
        .createAlias("seller", "s", JoinType.LEFT_OUTER_JOIN)
        .setFetchMode("s", FetchMode.JOIN)
        .list();

```

这个查询会返回所有的 `Item` 实例, 使用一个外联结加载 `Item#bids` 集合, 并且使用一个内联结加载 `Bid#bidder`。还会加载 `Item#seller`: 因为它不能为 `null`, 所以使用内联结还是

外联结没什么关系。一如往常，不要在一个查询中抓取几个集合，否则你会创建一个笛卡尔积(参阅 15.4.5 节)。

接下来，你会看到带有条件的子查询也适用于嵌套的 Criteria 实例。

16.3.5 子查询

以下子查询会返回正在售卖多个商品的所有 User 实例：

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
DetachedCriteria sq = DetachedCriteria.forClass(Item.class, "i");
sq.add(Restrictions.eqProperty("i.seller.id", "u.id"));
sq.setProjection(Projections.rowCount());
```

```
List<User> result =
    session.createCriteria(User.class, "u")
        .add(Subqueries.lt(11, sq))
        .list();
```

DetachedCriteria 是一个返回限制为由指定 User 售出的商品数量的查询。该限制依赖别名 u，因此这是一个相关子查询。然后“外”查询会嵌入 DetachedCriteria 并且提供别名 u。注意，该子查询是 lt() (小于)操作的右操作符，它在 SQL 中会转换成 $1 < ([\text{Result of count query}])$ 。

Hibernate 还支持量化。这个查询会返回所有出价都小于或等于 10 的商品：

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
DetachedCriteria sq = DetachedCriteria.forClass(Bid.class, "b");
sq.add(Restrictions.eqProperty("b.item.id", "i.id"));
sq.setProjection(Projections.property("amount"));
```

```
List<Item> result =
    session.createCriteria(Item.class, "i")
        .add(Subqueries.geAll(new BigDecimal(10), sq))
        .list();
```

同样，操作符的位置要求该比较要基于 geAll() (大于或等于所有)来找出金额“小于或等于 10”的出价。

到目前为止，只有很少的合理理由使用旧的 org.hibernate.Criteria API。不过，你真的应该在新的应用程序中使用标准化的 JPA 查询语言。我们已经介绍过，旧的专有 API 的最有意思的功能是在限制和投影中嵌入 SQL 表达式。另一个你可能认为有意思的仅用于 Hibernate 的功能是按例查询。

16.3.6 示例查询

示例查询背后的理念是，提供一个示例实体实例，而 Hibernate 会加载“看起来像该示例”的所有实体实例。如果用户界面中有一个复杂的搜索界面，那么这可能就很方便，因

为你不必编写额外的类来持有所输入的搜索词。

我们假设应用程序中有一个搜索表单，其中可以根据姓搜索 `User` 实例。可以将用于“姓”的表单字段直接绑定到 `User#lastname` 属性，然后告知 Hibernate 加载“类似的” `User` 实例：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
User template = new User();           ← ① 创建空的 User 实例
template.setLastname("Doe");

org.hibernate.criterion.Example example = Example.create(template); ← ② 创建 Example 实例
example.ignoreCase();
example.enableLike(MatchMode.START);
example.excludeProperty("activated");

List<User> users =
    session.createCriteria(User.class)
        .add(example)                  ← ④ 添加 Example 作为一个限制
        .list();
```

③ 忽略活动属性

① 创建 `User` 的一个“空”实例作为搜索的一个模板，并且设置要查找的属性值：姓为“Doe”的人。

② 用该模板创建 `Example` 的一个实例。这个 API 允许你调整搜索。你想要忽略姓的大小写，并且想要进行子字符串搜索，因此会匹配“Doe”、“DoeX”或“Doe Y”。

③ `User` 类有一个称为 `activated` 的 `boolean` 属性。作为一个基元，它不能为 `null`，并且其默认值为 `false`，因此 Hibernate 会将它包含在搜索中并且只返回非活动状态的用户。由于你想要所有的用户，因此要告知 Hibernate 忽略该属性。

④ 将 `Example` 添加到 `Criteria` 作为一个限制。

由于你已经遵循 `JavaBean` 规则编写了 `User` 实体类，因此将它绑定到一个 UI 表单应该非常简单。它具有普通的获取和设置方法，并且可以使用公共无参构造函数创建一个“空”实例(回想一下我们在 3.2.3 节中对于构造函数设计的探讨)。

`Example` API 的一个明显劣势在于，像 `ignoreCase()` 和 `enableLike()` 这样的任何字符串匹配选项都可用于模板的所有字符串值属性。如果同时搜索 `lastname` 和 `firstname`，那么这两者将都是大小写不敏感的子字符串匹配。

默认情况下，指定实体模板的所有非空值属性都会被添加到该示例查询的限制。正如上面的代码片段所示，可以使用 `excludeProperty()` 根据姓名手动排除实体模板的属性。其他的排除选项有，使用 `excludeZeroes()` 排除零值属性(比如 `int` 或 `long`)，以及使用 `excludeNone()` 完全禁用排除。如果未排除任何属性，就会使用 `is null` 检查将模板的所有 `null` 属性添加到 SQL 查询中的限制。

如果需要对属性排除和包含的更多控制，可以扩展 `Example` 并且编写你自己的 `PropertySelector`：

路径: `/examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java`

```
class ExcludeBooleanExample extends Example {
    ExcludeBooleanExample(Object template) {
```



```

super(template, new PropertySelector() {
    @Override
    public boolean include(Object propertyValue,
        String propertyName,
        Type type) {
        return propertyValue != null
            && !type.equals(StandardBasicTypes.BOOLEAN);
    }
});
}
}

```

这个选择器会排除所有 null 属性(就像默认选择器一样)并且追加排除所有 Boolean 属性(比如 User#activated)。

在将一个 Example 限制添加到 Criteria 之后,可以将进一步的限制添加到查询。或者,可以将多个示例限制添加到单个查询。以下查询会返回 name 以“B”或“b”开头的所有 Item 实例以及匹配一个 User 示例的 seller:

路径: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```

Item itemTemplate = new Item();
itemTemplate.setName("B");

Example exampleItem = Example.create(itemTemplate);
exampleItem.ignoreCase();
exampleItem.enableLike(MatchMode.START);
exampleItem.excludeProperty("auctionType");
exampleItem.excludeProperty("createdOn");

User userTemplate = new User();
userTemplate.setLastname("Doe");

Example exampleUser = Example.create(userTemplate);
exampleUser.excludeProperty("activated");

List<Item> items =
    session
        .createCriteria(Item.class)
        .add(exampleItem)
        .createCriteria("seller").add(exampleUser)
        .list();

```

此时,我们请你重新思考一下,使用手动编码的 SQL/JDBC 实现这样一个搜索会需要多少代码量。

16.4 本章小结

- 介绍了使用 ResultTransformer API 编写自定义代码以便处理查询结果,返回一系列列表以及一系列映射,并且将别名映射到 bean 属性。

- 介绍了 **Hibernate** 的集合过滤接口以及更好地使用映射的持久化集合。
- 探究了较老的 **Hibernate Criteria** 查询工具以及什么情况下可能使用它替代 **JPA** 中的标准化条件查询。我们介绍了使用此 **API** 的所有相关的和 **Hibernate** 的好处：选择和排序、限制、投影和聚合、联结、子查询以及示例查询。

自定义 SQL

第 17 章

17

本章内容简介:

- 回退到 JDBC
- 映射 SQL 查询结果
- 自定义 CRUD 操作
- 调用存储过程

在这一章中,我们会介绍在 Hibernate 应用程序中自定义以及嵌入 SQL。SQL 创建于 20 世纪 70 年代,但 ANSI 直到 1986 年才标准化它。尽管 SQL 标准的每次更新都提供了新的(以及许多有争议的)功能,但每一个支持 SQL 的 DBMS 产品都会以其自己独特的方式这样做。可移植性的重担再次落在了数据库应用程序开发人员身上。这就是 Hibernate 可以提供帮助的地方:其内置的查询语言可以生成依赖所配置数据库方言的 SQL。方言还有助于生成其他所有自动生成的 SQL(比如 Hibernate 必须按需检索一个集合时)。有了简单的方言切换,就可以在不同的 DBMS 上运行应用程序。Hibernate 会生成用于所有创建、读取、更新以及删除(CRUD)操作的全部 SQL 语句。

不过,有时需要的控制要比 Hibernate 和 Java 持久化 API 所提供的更多:需要处理较低级别的抽象。使用 Hibernate,可以编写 SQL 语句:

- 回退到 JDBC API,并且直接使用 Connection、PreparedStatement 以及 ResultSet 接口。Hibernate 提供了 Connection,因此不必维护一个单独的连接池,并且 SQL 语句会在相同的(当前)事务中执行。
- 编写普通的 SQL SELECT 语句,并且将它们嵌入到 Java 代码中或者(在 XML 文件或注解中)将它们外部化为命名查询。你要使用 Java 持久化 API 执行这些 SQL 查询,就像一个普通的 JPQL 查询一样。然后 Hibernate 可以根据映射来转换查询结果。这也适用于存储过程调用。
- 使用你自己手动编写的 SQL 替换 Hibernate 生成的 SQL 语句。例如,当 Hibernate 用 em.find()加载一个实体实例或者按需加载一个集合时,你自己的 SQL 查询就能执行该加载。还可以编写你自己的数据操作语言(DML)语句,比如 UPDATE、INSERT 和 DELETE。甚至可以调用一个存储过程来执行一个 CRUD 操作。可以使用自定义语句替换由 Hibernate 自动生成的所有 SQL 语句。

我们首先介绍 JDBC 的后备使用，然后探讨 Hibernate 的自动结果映射功能。之后，我们将介绍如何在 Hibernate 中重写查询和 DML 语句。最后，我们要探讨与数据库存储过程的集成。

JPA 2 中主要的新功能

- 可以将一个 SQL 查询结果映射到一个构造函数。
- 可以使用新的 StoredProcedureQuery API 直接调用存储过程和函数。

Hibernate 特性

17.1 回退到 JDBC

有时希望 Hibernate 不再成为阻碍并且通过 JDBC API 直接访问数据库。为此，需要一个 `java.sql.Connection` 接口来编写和执行 `PreparedStatement` 并且管理对语句 `ResultSet` 的访问。因为 Hibernate 已经知道如何获得以及关闭数据库连接，所以它能为应用程序提供一个 `Connection` 并且在完成时释放它。

`org.hibernate.jdbc.Work` API 提供了这一功能，一个回调形式的接口。要通过实现这个接口来封装 JDBC “工作”；Hibernate 会调用你提供一个 `Connection` 的实现。以下示例会执行一个 SQL `SELECT` 并且遍历 `ResultSet`。见代码清单 17.1。

代码清单 17.1 使用 JDBC 接口封装 “工作”

路径: /examples/src/test/java/org/jpwh/test/querying/sql/JDBCFallback.java

```
public class QueryItemWork implements org.hibernate.jdbc.Work {  
    final protected Long itemId; ← ❶ 商品标识符  
  
    public QueryItemWork(Long itemId) {  
        this.itemId = itemId;  
    }  
    ❷ 调用 execute()  
    @Override  
    public void execute(Connection connection) throws SQLException {  
        PreparedStatement statement = null;  
        ResultSet result = null;  
        try {  
            statement = connection.prepareStatement(  
                "select * from ITEM where ID = ?"  
            );  
            statement.setLong(1, itemId);  
            result = statement.executeQuery();  
            while (result.next()) {  
                String itemName = result.getString("NAME");  
                BigDecimal itemPrice = result.getBigDecimal("BUYNOWPRICE");  
                // ...  
            }  
        }  
    }  
}
```

```

    }
    } finally {
        if (result != null)
            result.close();
        if (statement != null)
            statement.close();
    }
}

```

← ③ 释放资源

- ❶ 对于这一“工作”，需要一个商品标识符，这是终态字段和构造函数参数强制要求的。
- ❷ `execute()` 方法是由 Hibernate 使用一个 JDBC Connection 调用的。无须在完成时关闭该连接。
- ❸ 不过，必须关闭并且释放你已经获得的其他资源，比如 `PreparedStatement` 和 `ResultSet`。

要使用 Hibernate Session API 执行该 Work：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/JDBCFallback.java

```

UserTransaction tx = TM.getUserTransaction();
tx.begin();
EntityManager em = JPA.createEntityManager();

Session session = em.unwrap(Session.class);
session.doWork(new QueryItemWork(ITEM_ID));
tx.commit();
em.close();

```

在这个例子中，Hibernate 已经调用了它为当前系统事务提供的 JDBC Connection。在该系统事务被提交时，就会提交你的语句，并且无论是否使用 `EntityManager` 或 `Session API` 执行的所有操作，都是相同工作单元的一部分。或者如果希望将 JDBC “工作” 的一个值返回给应用程序，则要实现接口 `org.hibernate.jdbc.ReturningWork`。

可以在一个 Work 实现中执行的 JDBC 操作并没有任何限制。可以使用一个 `CallableStatement` 而不是 `PreparedStatement` 执行数据库中的一个存储过程；你具有对 JDBC API 的完全访问。

对于简单查询以及使用 `ResultSet` 来说，比如上一个示例中的情况，可以使用更为便利的替代项。

17.2 映射 SQL 查询结果

当使用 JDBC API 执行一个 SQL SELECT 查询或者执行一个返回 `ResultSet` 的存储过程时，要遍历这个结果集的每一行并且检索需要的数据。这是一项劳动密集型任务，并且最终要反复复制相同行的代码。

快速测试 SQL 语句

要了解在不启动本地服务器的情况下使用几个 DBMS 测试 SQL 脚本的一种简单方式，可以查看 <http://sqlfiddle.com> 的在线服务 SQL Fiddle。

Hibernate 提供了一个可选项：执行原生 SQL 查询或者使用 Hibernate/Java 持久化 API 调用存储过程并且得到你所选的实例的一个 List，而非一个 ResultSet。可以将该 ResultSet 映射到任何你选择的类，并且 Hibernate 会为你执行该转换。

提示：在这一节中，我们只探讨了原生的 SELECT SQL 查询。还可以使用相同的 API 编写 UPDATE 和 INSERT 语句，我们将在 20.1 节中介绍这一点。

Hibernate 特性

如今，有两个 API 可用于执行原生 SQL 查询以及转换其结果：

- 用于嵌入 SQL 语句的具有 EntityManager#createNativeQuery() 的标准化 Java 持久化 API，以及用于外部化查询的 @NamedNativeQuery。可以使用 @SqlResultSetMapping 注解或在 JPA orm.xml 文件中映射结果集。还可以将命名的 SQL 查询外部化到 JPA XML 文件。
- 用于结果集映射的具有 Session#createSQLQuery() 和 org.hibernate.SQLQuery 的专有且较老的 Hibernate API。还可以在 Hibernate XML 元数据文件中定义外部化的命名的 SQL 查询和结果映射。

Hibernate API 还有更多的功能。例如，它 also 支持 SQL 结果映射中集合与实体关联的急加载。在后面几节中，你会看到这两个 API 用于每个查询的并列对比。我们首先介绍一个简单嵌入的 SQL 查询以及标量投影结果的映射。

17.2.1 使用 SQL 查询进行投影

以下查询会返回 Object[] 的一个 List，每个数组都表示 SQL 投影的一个元组(行)：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select NAME, AUCTIONEND from {h-schema}ITEM"
);
List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof String);
    assertTrue(tuple[1] instanceof Date);}
```

路径：/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select NAME, AUCTIONEND from {h-schema}ITEM"
);
List<Object[]> result = query.list();

for (Object[] tuple : result) {
```



```

    assertTrue(tuple[0] instanceof String);
    assertTrue(tuple[1] instanceof Date);
}

```

`em.createNativeQuery()` 和 `session.createSQLQuery()` 方法接受一个普通的 SQL 查询字符串。

该查询会检索 ITEM 表的 NAME 和 AUCTIONEND 列值，并且 Hibernate 会自动将它们映射为 String 和 `java.util.Date` 值。Hibernate 会读取 `java.sql.ResultSetMetaData` 来判定每个投影元素的类型，并且它知道 VARCHAR 列类型映射为一个 String，而 TIMESTAMP 映射为一个 `java.util.Date` (正如 5.3 节中所阐释的那样)。

Hibernate 的 SQL 查询引擎支持几种方便的占位符，比如前面示例中的 {h-schema}。Hibernate 会用持久化单元的默认模式 (`hibernate.default_schema` 配置属性) 替换此占位符。其他受支持的占位符用于默认 SQL 目录的 {h-catalog} 以及 {h-domain}，它会合并目录与模式值。

在 Hibernate 中执行一个 SQL 语句的最大优势是，将结果集自动封送处理到域模型 (实体) 类的实例中。

17.2.2 映射到一个实体类

这个 SQL 查询会返回 Item 实体实例的一个 List:

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java`

```

Query query = em.createNativeQuery(
    "select * from ITEM",
    Item.class
);

List<Item> result = query.getResultList();

```

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java`

```

org.hibernate.SQLQuery query = session.createSQLQuery(
    "select * from ITEM"
);

query.addEntity(Item.class);

List<Item> result = query.list();

```

所返回的 Item 实例处于持久化状态，由当前持久化上下文托管。因此该结果与使用 JPQL 查询 `select i from Item i` 所得到的结果相同。

为了此转换，Hibernate 会读取该 SQL 查询的结果集并且尝试找出在实体映射元数据中定义的列名和类型。如果返回了 AUCTIONEND 列，并且它被映射到 `Item#auctionEnd` 属性，那么 Hibernate 就会知道如何填充该属性并且返回完全加载的实体实例。

注意，Hibernate 会预期该查询返回创建一个 Item 实例所需的所有列，其中包括所有的属性、嵌入的组件以及外键列。如果 Hibernate 无法在结果集中找到一个映射的列 (根据名称)，则会抛出一个异常。你可能必须在 SQL 中使用别名来返回相同的在实体映射元数据中定义的列名。

接口 `javax.persistence.Query` 和 `org.hibernate.SQLQuery` 都支持参数绑定。以下查询仅会返回单个 `Item` 实体实例：

路径：`/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java`

```
Query query = em.createNativeQuery(
    "select * from ITEM where ID = ?",
    Item.class
);
query.setParameter(1, ITEM_ID);           ← 从 1 开始
List<Item> result = query.getResultList();
```

路径：`/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java`

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select * from ITEM where ID = ?"
);
query.addEntity(Item.class);
query.setParameter(0, ITEM_ID);           ← 从 0 开始
List<Item> result = query.list();
```

由于历史原因，Hibernate 会从 0 开始计算位置参数的数量，而 JPA 从 1 开始。命名参数的绑定通常更为健壮：

路径：`/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java`

```
Query query = em.createNativeQuery(
    "select * from ITEM where ID = :id",
    Item.class
);
query.setParameter("id", ITEM_ID);
List<Item> result = query.getResultList();
```

路径：`/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java`

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select * from ITEM where ID = :id"
);
query.addEntity(Item.class);
query.setParameter("id", ITEM_ID);
List<Item> result = query.list();
```

尽管 Hibernate 中的这两个 API 都可用，但 JPA 规范没有考虑将命名参数绑定用于原生查询移植。因此有些 JPA 提供程序可能不支持将命名参数用于原生查询。

如果 SQL 查询未将列作为 Java 实体类中的映射来返回，并且不能用别名重命名结果中的列来重写查询，那么就必须创建一个结果集映射。

17.2.3 自定义结果映射

以下查询会返回托管 Item 实体实例的一个 List。ITEM 表的所有列都包含在 SQL 投影中，正如一个 Item 实例的构造所需的那样。但该查询使用投影中的一个别名将 NAME 列重命名为 EXTENDED_NAME:

路径: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select " +
        "i.ID, " +
        "'Auction: ' || i.NAME as EXTENDED_NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID " +
        "from ITEM i",
    "ItemResult"
);

List<Item> result = query.getResultList();
```

Hibernate 不再能自动将结果集字段匹配到 Item 属性: 结果集中缺失了 NAME 列。因此要使用 createNativeQuery() 的第二个参数来指定一个“结果映射”, 此处是 ItemResult。

将结果字段映射到实体属性

例如, 可以使用 Item 类上的注解来声明此映射:

路径: /model/src/main/java/org/jpwh/model/querying/Item.java

```
@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "ItemResult",
        entities =
            @EntityResult(
                entityClass = Item.class,
                fields = {
                    @FieldResult(name = "id", column = "ID"),
                    @FieldResult(name = "name", column = "EXTENDED_NAME"),
                    @FieldResult(name = "createdOn", column = "CREATEDON"),
                    @FieldResult(name = "auctionEnd", column = "AUCTIONEND"),
                    @FieldResult(name = "auctionType", column = "AUCTIONTYPE"),
                    @FieldResult(name = "approved", column = "APPROVED"),
                    @FieldResult(name = "buyNowPrice", column = "BUYNOWPRICE"),
                    @FieldResult(name = "seller", column = "SELLER_ID")
                }
            )
    )
})
```



```

    })
    @Entity
    public class Item {
        // ...
    }

```

要将结果集的所有字段映射到实体类的属性。即便只有一个字段/列不匹配已经映射的列名称(此处是 EXTENDED_NAME), 所有其他的列和属性也都必须被映射。

注解中映射的 SQL 结果难以被读取, 并且像使用 JPA 注解一样, 它们仅在声明在一个类上时可用, 而非在 package-info.java 元数据文件中可用。我们更愿意将这样的映射外部化到 XML 文件中。以下 XML 片段提供了相同的映射:

路径: /model/src/main/resources/querying/NativeQueries.xml

```

<sql-result-set-mapping name="ExternalizedItemResult">
    <entity-result entity-class="org.jpwh.model.querying.Item">
        <field-result name="id" column="ID"/>
        <field-result name="name" column="EXTENDED_NAME"/>
        <field-result name="createdOn" column="CREATEDON"/>
        <field-result name="auctionEnd" column="AUCTIONEND"/>
        <field-result name="auctionType" column="AUCTIONTYPE"/>
        <field-result name="approved" column="APPROVED"/>
        <field-result name="buyNowPrice" column="BUYNOWPRICE"/>
        <field-result name="seller" column="SELLER_ID"/>
    </entity-result>
</sql-result-set-mapping>

```

如果这两个结果集映射都具有相同的名称, 那么在 XML 中声明的映射就会重写用注解定义的映射。

还可以使用 @NamedNativeQuery 或 <named-native-query> 外部化实际的 SQL 查询, 如 14.4 节中所示。在所有以下示例中, 我们都将 SQL 语句保持在 Java 代码中嵌入, 因为这会让你更易于理解该代码的功能。但大多数时候, 你会看到使用更简明的 XML 语法的结果集映射。

我们首先用专有的 Hibernate API 重复上一个查询:

路径: /examples/src/test/java/org.jpwh/test/querying/sql/HibernateSQLQueries.java

```

org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "i.ID as {i.id}, " +
        "'Auction: ' || i.NAME as {i.name}, " +
        "i.CREATEDON as {i.createdOn}, " +
        "i.AUCTIONEND as {i.auctionEnd}, " +
        "i.AUCTIONTYPE as {i.auctionType}, " +
        "i.APPROVED as {i.approved}, " +
        "i.BUYNOWPRICE as {i.buyNowPrice}, " +
        "i.SELLER_ID as {i.seller} " +
        "from ITEM i"
);

```

```
query.addEntity("i", Item.class);
```

```
List<Item> result = query.list();
```

使用 Hibernate API，可以通过别名占位符在该查询内部直接执行结果集映射。在调用 `addEntity()` 时，要提供一个别名，这里是 `i`。然后在 SQL 字符串中，要让 Hibernate 用 `{i.name}` 和 `{i.auctionEnd}` 这样的占位符生成投影中的实际别名，它们指向的是 `Item` 实体的属性。无须额外的结果集映射声明；Hibernate 会生成 SQL 字符串中的别名并且知道如何从查询 `ResultSet` 中读取属性值。这比 JPA 结果集映射选项更为方便。

或者，如果不能或不希望修改 SQL 语句，则使用 `org.hibernate.SQLQuery` 上的 `addRoot()` 和 `addProperty()` 来执行该映射：

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java`

```
org.hibernate.SQLQuery query = session.createSQLQuery(
```

```
    "select " +
```

```
        "i.ID, " +
```

```
        "'Auction: ' || i.NAME as EXTENDED_NAME, " +
```

```
        "i.CREATEDON, " +
```

```
        "i.AUCTIONEND, " +
```

```
        "i.AUCTIONTYPE, " +
```

```
        "i.APPROVED, " +
```

```
        "i.BUYNOWPRICE, " +
```

```
        "i.SELLER_ID " +
```

```
        "from ITEM i"
```

```
);
```

```
query.addRoot("i", Item.class)
```

```
    .addProperty("id", "ID")
```

```
    .addProperty("name", "EXTENDED_NAME")
```

```
    .addProperty("createdOn", "CREATEDON")
```

```
    .addProperty("auctionEnd", "AUCTIONEND")
```

```
    .addProperty("auctionType", "AUCTIONTYPE")
```

```
    .addProperty("approved", "APPROVED")
```

```
    .addProperty("buyNowPrice", "BUYNOWPRICE")
```

```
    .addProperty("seller", "SELLER_ID");
```

```
List<Item> result = query.list();
```

就像使用标准 API 一样，还可以使用 Hibernate API 借助按名称的已有结果集映射：

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java`

```
org.hibernate.SQLQuery query = session.createSQLQuery(
```

```
    "select " +
```

```
        "i.ID, " +
```

```
        "'Auction: ' || i.NAME as EXTENDED_NAME, " +
```

```
        "i.CREATEDON, " +
```

```
        "i.AUCTIONEND, " +
```

```
        "i.AUCTIONTYPE, " +
```

```
        "i.APPROVED, " +
```

```
        "i.BUYNOWPRICE, " +
```

```

        "i.SELLER_ID " +
        "from ITEM i"
    );
    query.setResultSetMapping("ItemResult");

```

```
List<Item> result = query.list();
```

另一个需要自定义结果集映射的例子是 SQL 查询结果集中的重复列名。

映射重复字段

以下查询会在单个语句中联结 ITEM 和 USERS 表来加载每个 Item 的 seller:

路径: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```

Query query = em.createNativeQuery(
    "select " +
        "i.ID as ITEM_ID, " +
        "i.NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID, " +
        "u.ID as USER_ID, " +
        "u.USERNAME, " +
        "u.FIRSTNAME, " +
        "u.LASTNAME, " +
        "u.ACTIVATED, " +
        "u.STREET, " +
        "u.ZIPCODE, " +
        "u.CITY " +
        "from ITEM i join USERS u on u.ID = i.SELLER_ID",
    "ItemSellerResult"
);
List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof Item);
    assertTrue(tuple[1] instanceof User);
    Item item = (Item) tuple[0];
    assertTrue(Persistence.getPersistenceUtil().isLoaded(item, "seller"));
    assertEquals(item.getSeller(), tuple[1]);
}

```

这实际上是关联 Item#seller 的急抓取。Hibernate 知道每个行都包含一个 Item 和一个 User 实体实例的字段，由 SELLER_ID 链接。

该结果集中的重复列就是 i.ID 和 u.ID，它们都具有相同的名称。已经用别名将它们重命名为 ITEM_ID 和 USER_ID，这样就必须映射如何转换该结果集：

路径: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="ItemSellerResult">
  <entity-result entity-class="org.jpwh.model.querying.Item">
    <field-result name="id" column="ITEM_ID"/>
    <field-result name="name" column="NAME"/>
    <field-result name="createdOn" column="CREATEDON"/>
    <field-result name="auctionEnd" column="AUCTIONEND"/>
    <field-result name="auctionType" column="AUCTIONTYPE"/>
    <field-result name="approved" column="APPROVED"/>
    <field-result name="buyNowPrice" column="BUYNOWPRICE"/>
    <field-result name="seller" column="SELLER_ID"/>
  </entity-result>
  <entity-result entity-class="org.jpwh.model.querying.User">
    <field-result name="id" column="USER_ID"/>
    <field-result name="name" column="NAME"/>
    <field-result name="username" column="USERNAME"/>
    <field-result name="firstname" column="FIRSTNAME"/>
    <field-result name="lastname" column="LASTNAME"/>
    <field-result name="activated" column="ACTIVATED"/>
    <field-result name="homeAddress.street" column="STREET"/>
    <field-result name="homeAddress.zipcode" column="ZIPCODE"/>
    <field-result name="homeAddress.city" column="CITY"/>
  </entity-result>
</sql-result-set-mapping>
```

像之前一样, 必须将每个实体结果的所有字段映射到列名, 即便与原始实体映射相比只有两个字段有不同的名称。

这个查询更易于使用 Hibernate API 来映射:

路径: /examples/src/test/java/org.jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "{i.*}, {u.*} " +
        "from ITEM i join USERS u on u.ID = i.SELLER_ID"
);
query.addEntity("i", Item.class);
query.addEntity("u", User.class);

List<Object[]> result = query.list();
```

Hibernate 会为 {i.*} 和 {u.*} 占位符将自动生成的唯一别名添加到 SQL 语句, 因此该查询不会返回重复的列名。

你可能已经注意到了上一个 JPA 结果映射中用于 User 内 homeAddress 嵌入组件的圆点语法。我们再次看看这个特殊情况。

将字段映射到组件属性

User 有一个 homeAddress, 它是 Address 类的一个嵌入实例。以下查询会加载所有的 User 实例:

路径: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select " +
        "u.ID, " +
        "u.USERNAME, " +
        "u.FIRSTNAME, " +
        "u.LASTNAME, " +
        "u.ACTIVATED, " +
        "u.STREET as USER_STREET, " +
        "u.ZIPCODE as USER_ZIPCODE, " +
        "u.CITY as USER_CITY " +
        "from USERS u",
    "UserResult"
);

List<User> result = query.getResultList();
```

在这个查询中,重命名了 STREET、ZIPCODE 和 CITY 列,因此必须手动将它们映射到嵌入的组件属性:

路径: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="UserResult">
    <entity-result entity-class="org.jpwh.model.querying.User">
        <field-result name="id" column="ID"/>
        <field-result name="name" column="NAME"/>
        <field-result name="username" column="USERNAME"/>
        <field-result name="firstname" column="FIRSTNAME"/>
        <field-result name="lastname" column="LASTNAME"/>
        <field-result name="activated" column="ACTIVATED"/>
        <field-result name="homeAddress.street" column="USER_STREET"/>
        <field-result name="homeAddress.zipcode" column="USER_ZIPCODE"/>
        <field-result name="homeAddress.city" column="USER_CITY"/>
    </entity-result>
</sql-result-set-mapping>
```

之前在探讨嵌入组件时已经介绍过几次此圆点语法:要使用 `homeAddress.street` 来引用 `homeAddress` 的 `street` 属性。对于嵌套的嵌入组件,如果 `City` 不只是一个字符串而是另一个可嵌入类的话,就可以编写 `homeAddress.city.name`。

Hibernate 的 SQL 查询 API 还支持用于组件属性的别名占位符中的圆点语法。这里是相同的查询和结果集映射:

路径: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "u.ID as {u.id}, " +
        "u.USERNAME as {u.username}, " +
        "u.FIRSTNAME as {u.firstname}, " +
        "u.LASTNAME as {u.lastname}, " +
```

```

        "u.ACTIVATED as {u.activated}, " +
        "u.STREET as {u.homeAddress.street}, " +
        "u.ZIPCODE as {u.homeAddress.zipcode}, " +
        "u.CITY as {u.homeAddress.city} " +
        "from USERS u"
    );
    query.addEntity("u", User.class);

```

```
List<User> result = query.list();
```

用一个 SQL 查询急抓取集合仅在使用 Hibernate API 时才可用。

Hibernate 特性

急抓取集合

我们假设你希望使用一个 SQL 查询加载所有的 Item 实例并且同时初始化每个 Item 的 bids 集合。这就要求在 SQL 查询中使用一个外联结：

路径: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```

org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "i.ID as ITEM_ID, " +
        "i.NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID, " +
        "b.ID as BID_ID, " +
        "b.ITEM_ID as BID_ITEM_ID, " +
        "b.AMOUNT, " +
        "b.BIDDER_ID " +
        "from ITEM i left outer join BID b on i.ID = b.ITEM_ID"
);
query.addRoot("i", Item.class)
    .addProperty("id", "ITEM_ID")
    .addProperty("name", "NAME")
    .addProperty("createdOn", "CREATEDON")
    .addProperty("auctionEnd", "AUCTIONEND")
    .addProperty("auctionType", "AUCTIONTYPE")
    .addProperty("approved", "APPROVED")
    .addProperty("buyNowPrice", "BUYNOWPRICE")
    .addProperty("seller", "SELLER_ID");
query.addFetch("b", "i", "bids")
    .addProperty("key", "BID_ITEM_ID")
    .addProperty("element", "BID_ID")
    .addProperty("element.id", "BID_ID")
    .addProperty("element.item", "BID_ITEM_ID")

```

① 联结 ITEM 和 BID

② 将列映射到实体属性

③ 将 Bid 属性映射到结果集


```

        .addProperty("element.amount", "AMOUNT")
        .addProperty("element.bidder", "BIDDER_ID");

List<Object[]> result = query.list();
assertEquals(result.size(), 5);

for (Object[] tuple : result) {
    Item item = (Item) tuple[0];
    assertTrue(Persistence.getPersistenceUtil().isLoaded(item, "bids"));

    Bid bid = (Bid) tuple[1];
    if (bid != null)
        assertTrue(item.getBids().contains(bid));
}

```

④ 结果中有 5 行

⑤ 第一个结果是 Item 实例

⑥ 第二个结果是每个 Bid

- ❶ 该查询(外)联结了 ITEM 和 BID 表。该投影会将需要的所有列返回到构造 Item 和 Bid 实例。该查询使用别名重命名了像 ID 这样的重复列，因此字段名在结果中唯一的。
- ❷ 由于这些重命名的字段，必须将每个列映射到其对应的实体属性。
- ❸ 使用所属实体 i 的别名为 bids 集合添加一个 FetchReturn，并且将 key 和 element 特殊属性映射到外键列 BID_ITEM_ID 以及 Bid 的标识符。然后该代码会将 Bid 的每个属性映射到结果集的一个字段。有些字段会被映射两次，这是因为 Hibernate 需要用于集合的构造。
- ❹ 结果集中的行数是一个积：一个商品有三次出价、一个商品有一次出价，并且最后一个商品没有出价，结果中总共有五行。
- ❺ 结果元组的第一个元素是 Item 实例；Hibernate 初始化该出价集合。
- ❻ 结果元组的第二个元素是每一个 Bid。

另外，如果因为由 SQL 查询返回的字段名匹配已经映射的实体列，而不必手动映射该结果，那么就能让 Hibernate 用占位符将别名插入 SQL 语句中：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```

org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "{i.*}, " +
        "{b.*} " +
        "from ITEM i left outer join BID b on i.ID = b.ITEM_ID"
);
query.addEntity("i", Item.class);
query.addFetch("b", "i", "bids");

List<Object[]> result = query.list();

```

使用动态 SQL 结果映射急抓取集合只在使用 Hibernate API 时才可用；它并不是在 JPA 中标准化的。

使用 SQL 查询抓取集合的限制

使用 `org.hibernate.SQLQuery` API, 就只能抓取实体关联的一个集合; 即, 一对多或多对多集合。在编写本书时, Hibernate 还不支持用专用动态映射将 SQL 结果集映射到基本或可嵌入类型的集合。比如, 这意味着你不能用一个自定义 SQL 查询和 `org.hibernate.SQLQuery` API 急加载 `Item#images` 集合。

到目前为止, 你已经看到了 SQL 查询返回托管的实体实例。还可以使用正确的构造函数返回任何类的瞬时实例。

将结果映射到一个构造函数

我们在 15.3.2 节中介绍了在一个查询中使用 JPQL 和条件示例的动态实例化。JPA 支持使用原生查询的相同功能。以下查询会返回 `ItemSummary` 实例的一个 List:

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java`

```
Query query = em.createNativeQuery(
    "select ID, NAME, AUCTIONEND from ITEM",
    "ItemSummaryResult"
);
List<ItemSummary> result = query.getResultList();
```

`ItemSummaryResult` 映射会将查询结果的每个列转换成用于 `ItemSummary` 构造函数的一个参数:

路径: `/model/src/main/resources/querying/NativeQueries.xml`

```
<sql-result-set-mapping name="ItemSummaryResult">
  <constructor-result target-class="org.jpwh.model.querying.ItemSummary">
    <column name="ID" class="java.lang.Long"/>
    <column name="NAME"/>
    <column name="AUCTIONEND"/>
  </constructor-result>
</sql-result-set-mapping>
```

所返回的列类型必须匹配构造函数参数类型; Hibernate 会默认为 ID 列的 `BigInteger`, 因此你要将它映射到具有该类属性的 `Long`。

Hibernate API 为你提供了一个选择。可以使用用于按名称查询的已有结果映射, 或者应用一个结果转换器, 就像你在 16.1 节中所看到的为 JPQL 查询所做的那样:

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java`

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select ID, NAME, AUCTIONEND from ITEM"
```

```
);
```

```
// query.setResultSetMapping("ItemSummaryResult");
```

① 使用已有的结果映射

```
query.addScalar("ID", StandardBasicTypes.LONG);
```

```
query.addScalar("NAME");
```

② 将字段映射为标量

```

query.addScalar("AUCTIONEND");

query.setResultTransformer(
    new AliasToBeanConstructorResultTransformer(
        ItemSummary.class.getConstructor(
            Long.class,
            String.class,
            Date.class
        )
    )
);

List<ItemSummary> result = query.list();

```

③应用结果转换器

- ① 可以使用一个已有结果映射。
- ② 或者，可以将查询返回的字段映射为标量值。在不使用一个结果转换器的情况下，会得到用于每个结果行的一个 `Object[]`。
- ③ 应用一个内置结果转换器来将 `Object[]` 转换成 `ItemSummary` 的实例。
正如在 15.3.2 节中所阐释的，`Hibernate` 可以用这样的映射来使用任何类构造函数。可以构造 `Item` 实例而不是 `ItemSummary`。它们要么处于瞬时状态，要么处于分离状态，这取决于你是否在查询中返回并且映射一个标识符值。
还可以混合不同种类的结果映射或者直接返回标量值。

标量和混合的结果映射

接下来的查询会返回 `Object[]` 的一个 `List`，其中第一个元素是一个 `Item` 实体实例，而第二个元素是反射用于该商品的出价数的一个标量：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```

Query query = em.createNativeQuery(
    "select " +
        "i.*, " +
        "count(b.ID) as NUM_OF_BIDS " +
        "from ITEM i left join BID b on b.ITEM_ID = i.ID " +
        "group by i.ID, i.NAME, i.CREATEDON, i.AUCTIONEND, " +
        "i.AUCTIONTYPE, i.APPROVED, i.BUYNOWPRICE, i.SELLER_ID",
    "ItemBidResult"
);
List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof Item);
    assertTrue(tuple[1] instanceof Number);
}

```

该结果映射很简单，因为该投影不包含重复列：

路径：/model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="ItemBidResult">
```



```

<entity-result entity-class="org.jpwh.model.querying.Item"/>
<column-result name="NUM_OF_BIDS"/>
</sql-result-set-mapping>

```

使用 Hibernate API，就会使用 `addScalar()` 映射额外的标量结果：

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java`

```

org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "i.*, " +
        "count(b.ID) as NUM_OF_BIDS " +
        "from ITEM i left join BID b on b.ITEM_ID = i.ID " +
        "group by i.ID, i.NAME, i.CREATEDON, i.AUCTIONEND, " +
        "i.AUCTIONTYPE, i.APPROVED, i.BUYNOWPRICE, i.SELLER_ID"
);
query.addEntity(Item.class);
query.addScalar("NUM_OF_BIDS");

List<Object[]> result = query.list();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof Item);
    assertTrue(tuple[1] instanceof Number);
}

```

最后，在单个结果映射中，可以合并实体、构造函数以及标量列结果。以下查询会返回一个持久化和托管的 `User` 实体实例，它是同时返回的 `ItemSummary` 的 `seller`。你还会得到每个商品的出价次数：

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java`

```

Query query = em.createNativeQuery(
    "select " +
        "u.*, " +
        "i.ID as ITEM_ID, i.NAME as ITEM_NAME, i.AUCTIONEND as " +
        "ITEM_AUCTIONEND, " +
        "count(b.ID) as NUM_OF_BIDS " +
        "from ITEM i " +
        "join USERS u on u.ID = i.SELLER_ID " +
        "left join BID b on b.ITEM_ID = i.ID " +
        "group by u.ID, u.USERNAME, u.FIRSTNAME, u.LASTNAME, " +
        "u.ACTIVATED, u.STREET, u.ZIPCODE, u.CITY, " +
        "ITEM_ID, ITEM_NAME, ITEM_AUCTIONEND",
    "SellerItemSummaryResult"
);

List<Object[]> result = query.getResultList();
for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof User);
    assertTrue(tuple[1] instanceof BigInteger);
}

```

结果的错误顺序: Hibernate
问题 HHH-8678

```

    assertTrue(tuple[2] instanceof ItemSummary);
}

```

这个查询的结果映射如下：

路径：/model/src/main/resources/querying/NativeQueries.xml

```

<sql-result-set-mapping name="SellerItemSummaryResult">
    <entity-result entity-class="org.jpwh.model.querying.User"/>
    <constructor-result target-class="org.jpwh.model.querying.ItemSummary">
        <column name="ID" class="java.lang.Long"/>
        <column name="ITEM_NAME"/>
        <column name="ITEM_AUCTIONEND"/>
    </constructor-result>
    <column-result name="NUM_OF_BIDS"/>
</sql-result-set-mapping>

```

JPA 规范会确保在混合的结果映射中，所生成的每个元组的 `Object[]` 都会包含以下顺序的元素：首先是所有的 `<entity-result>` 数据，然后是 `<constructor-result>` 数据，最后是 `<column-result>` 数据。JPA XML 架构会强制使用结果映射声明中的这个顺序；但即便你用注解以不同顺序映射元素（它不能强制使用你声明映射的顺序），该查询结果也会使用标准顺序。注意，正如代码示例中所示的，在编写本书时，Hibernate 不会返回正确的顺序。

注意，使用 Hibernate 查询 API 时，可以使用相同的按名称的结果映射，就像我们之前介绍的那样。如果需要对结果封送处理的更多程式化控制，就必须编写你自己的结果转换器，因为没有内置的转换器可以自动映射这样的混合查询结果。

最后，你会看到使用 XML 文件中声明的 SQL 查询的一个更为复杂的示例。

17.2.4 外部化原生查询

现在我们介绍如何在 XML 文件中声明一个 SQL 查询，以替代在代码中嵌入字符串。在使用长 SQL 语句的真实应用程序中，阅读 Java 代码中的连串字符串会令人不快，因此你几乎总是会选择使用 XML 文件中的 SQL 语句。这也简化了专用的测试，因为可以在 XML 文件和 SQL 数据库控制台之间复制和粘贴 SQL 语句。

你大概已经注意到了，前面几节中的所有 SQL 示例都很平常。实际上，这些示例都不需要用 SQL 编写一个查询——我们可以在每个例子中使用 JPQL。为了让接下来的示例更有趣，我们要编写一个仅能用 SQL 而不能用 JPQL 表述的查询。

目录树

思考 Category 类及其自引用多对一关联，如图 17-1 所示。

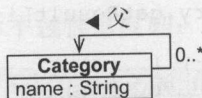


图 17-1 Category 具有一个自引用多对一关联

这是该关联的映射——PARENT_ID 外键列的一个普通@ManyToOne:

路径: /model/src/main/java/org/jpwh/model/querying/Category.java

```
@Entity
public class Category {

    @ManyToOne
    @JoinColumn(
        name = "PARENT_ID",
        foreignKey = @ForeignKey(name = "FK_CATEGORY_PARENT_ID")
    )
    protected Category parent;
    // ...
}
```

树的根节点没有父节点; 列必须可为空

目录形成了一棵树。这棵树的根是一个没有父节点的 Category 节点。图 17-2 中是该示例树的数据库数据。

CATEGORY		
ID	NAME	PARENT_ID
1	One	
2	Two	1
3	Three	1
4	Four	2

图 17-2 用于一个目录树的数据库表和样本数据

还可以将这些数据表示为一个树图，如图 17-3 所示。或者，可以使用一系列路径以及每个节点的级别:

```
/One, 0
/One/Two, 1
/One/Three, 1
/One/Two/Four, 2
```

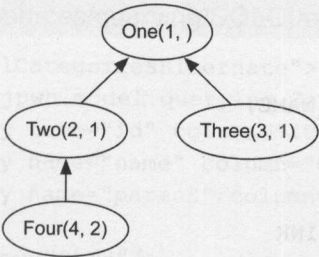


图 17-3 目录树

现在，思考你的应用程序如何加载 Category 实例。你可能希望找到这棵树的根 Category。这是一个普通的 JPQL 查询:

```
select c from Category c where c.parent is null
```


可以轻易查询这棵树特定级别的目录，比如根的所有子节点：

```
select c from Category c, Category r where r.parent is null and c.parent = r
```

这个查询只会返回根的直接子节点：此处是目录 Two 和 Three。

如何才能在一个查询中加载整棵树(或一棵子树)? 使用 JPQL 无法达成该目的, 因为它需要递归: “加载这一级别的目录, 然后加载下一级别的所有子节点, 然后加载那些子节点的所有子节点, 以此类推。” 在 SQL 中, 可以使用公用表表达式(common table expression, CTE)编写这样一个查询, 这个功能也称为子查询因式分解。

加载树

以下 SQL 查询会加载 JPA XML 文件中声明的 Category 实例的整棵树：

路径: /model/src/main/resources/querying/NativeQueries.xml

```
<named-native-query name="findAllCategories"
    result-set-mapping="CategoryResult">
```

```
<query>
```

```
    with CATEGORY_LINK(ID, NAME, PARENT_ID, PATH, LEVEL) as (
```

```
        select
```

```
            ID,
```

```
            NAME,
```

```
            PARENT_ID,
```

```
            '/' || NAME,
```

```
            0
```

```
        from CATEGORY where PARENT_ID is null
```

```
        union all
```

```
        select
```

```
            c.ID,
```

```
            c.NAME,
```

```
            c.PARENT_ID,
```

```
            cl.PATH || '/' || c.NAME,
```

```
            cl.LEVEL + 1
```

```
        from CATEGORY_LINK cl
```

```
        join CATEGORY c on cl.ID = c.PARENT_ID
```

```
    )
```

```
    select
```

```
        ID,
```

```
        NAME as CAT_NAME,
```

```
        PARENT_ID,
```

```
        PATH,
```

```
        LEVEL
```

```
    from CATEGORY_LINK
```

```
    order by ID
```

```
</query>
```

```
</named-native-query>
```

它是一个复杂查询, 为了理解它, 你要阅读最后一个 SELECT, 它会查询 CATEGORY_LINK 视图。该视图中的每一行都代表树中的一个节点。此处是在 WITH() AS 操作中声明

该视图的。CATEGORY_LINK 视图是其他两个 SELECT 的合并(联合)结果。你要在递归期间将额外的信息添加到视图，比如每个节点的 PATH 和 LEVEL。

我们来映射这个查询的结果：

路径：/model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="CategoryResult">
  <entity-result entity-class="org.jpwh.model.querying.Category">
    <field-result name="id" column="ID"/>
    <field-result name="name" column="CAT_NAME"/>
    <field-result name="parent" column="PARENT_ID"/>
  </entity-result>
  <column-result name="PATH"/>
  <column-result name="LEVEL" class="java.lang.Integer"/>
</sql-result-set-mapping>
```

XML 会将 ID、CAT_NAME 和 PARENT_ID 字段映射到 Category 实体的属性。该映射会将 PATH 和 LEVEL 作为额外的标量值来返回。

要执行命名的 SQL 查询并且访问结果，可以编写以下代码：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNamedQuery("findAllCategories");
List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    Category category = (Category) tuple[0];
    String path = (String) tuple[1];
    Integer level = (Integer) tuple[2];
    // ...
}
```

每个元组都包含一个托管的、持久化 Category 实例；它在树中的形如一个字符串(比如/One、/One/Two 等)的路径；以及该节点的树级别。

或者，可以在 Hibernate XML 元数据文件中声明和映射一个 SQL 查询：

路径：/model/src/main/resources/querying/SQLQueries.hbm.xml

```
<sql-query name="findAllCategoriesHibernate">
  <return class="org.jpwh.model.querying.Category">
    <return-property name="id" column="ID"/>
    <return-property name="name" column="CAT_NAME"/>
    <return-property name="parent" column="PARENT_ID"/>
  </return>
  <return-scalar column="PATH"/>
  <return-scalar column="LEVEL" type="integer"/>
  ...
</sql-query>
```

我们不探讨这个代码段中的 SQL 查询；它与之前在 JPA 示例中介绍的 SQL 语句相同。正如 14.4 节中介绍的，就 Java 代码中的执行而言，你用什么语法在 XML 文件或注解

中声明你的命名查询并不重要。什么语言也不重要——它可以是 JPQL 或 SQL。Hibernate 和 JPA 查询接口都有方法来“得到一个命名查询”并且执行它与你定义它的方式无关。

到此就完成了我们对于查询的 SQL 结果映射的探讨。下一个主题是用于 CRUD 操作的 SQL 语句的自定义，以替代 Hibernate 为在数据库中创建、读取、更新和删除数据所自动生成的 SQL。

Hibernate 特性

17.3 自定义 CRUD 操作

你编写的第一个自定义 SQL 会加载 User 类的一个实体实例。所有后续代码示例都显示了 Hibernate 默认会自动执行的相同 SQL，没有太多自定义——这有助于你更快速地理解映射技术。

可以用一个加载器来自定义实体实例的检索。

17.3.1 启用自定义加载器

当重写一个 SQL 查询来加载一个实体实例时，Hibernate 有两个要求：

- 编写一个检索实体实例的命名查询。我们介绍了一个 SQL 中的示例，但像往常一样，还可以在 JPQL 中编写命名查询。对于一个 SQL 查询，你可能需要一个自定义结果映射，就像本章之前内容所示。
- 在实体类上使用@org.hibernate.annotations.Loader 激活查询。这会将该查询启用为 Hibernate 所生成查询的替代。

我们来重写 Hibernate 加载 User 实体实例的方式，如代码清单 17.2 所示。

代码清单17.2 用一个自定义查询加载一个User实例
路径：/model/src/main/java/org/jpwh/model/customsql/User.java

```
@NamedNativeQueries({                                ← ❶声明查询以加载 User
    @NamedNativeQuery(
        name = "findUserById",                        ← ❷参数占位符
        query = "select * from USERS where ID = ?",
        resultClass = User.class                      ← ❸不需要自定义映射
    )
})
@org.hibernate.annotations.Loader(                  ← ❹将加载器设置为命名的查询
    namedQuery = "findUserById"
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```


- ❶ 注解声明该查询来加载一个 User 实例；还可以在 XML 文件(JPA 或 Hibernate 元数据)中声明它。可以在需要时在数据访问代码中直接调用此命名查询。
- ❷ 该查询必须正好具有一个参数占位符，Hibernate 会将其设置为要加载的实例的标识符值。这里它是一个位置参数，但一个命名参数也是适用的。
- ❸ 对于此普通查询，不需要一个自定义结果集映射。User 类会映射该查询所返回的所有字段。Hibernate 可以自动转换该结果。
- ❹ 将用于一个实体类的加载器设置为一个命名查询，这个命名查询会启用用于所有从数据库检索 User 实例的操作的查询。这里没有该查询语言或你要在何处声明它的提示；这是独立于加载器声明的。

在用于实体的一个命名加载器查询中，必须 SELECT 实体类的以下属性(即为之执行一个投影)：

- 如果使用了一个复合主键，则必须使用一个或多个标识符属性的值。
- 所有基本类型的标量属性。
- 嵌入组件的所有属性。
- 一个用于每个映射实体关联的每个 @JoinColumn 的实体标识符值，比如加载的实体类所拥有的 @ManyToOne。
- 位于 @SecondaryTable 注解中的所有标量属性、嵌入的组件属性以及关联联结引用。
- 如果通过拦截和字节码指令为某些属性启用延迟加载，就无须加载该延迟属性(参阅 12.1.3 节)。

Hibernate 总是会在必须根据标识符从数据库中检索一个 User 时调用该启用的加载器查询。当调用 `em.find(User.class, USER_ID)` 时，你的自定义查询将会执行。当调用 `someItem.getSeller().getUsername()` 时，`Item#seller` 代理必须被初始化，你的自定义查询将加载该数据。

你也可能希望自定义 Hibernate 如何在数据库中创建、更新和删除一个 User 实例。

17.3.2 自定义创建、更新和删除

Hibernate 通常会在启动时生成 CRUD SQL 语句。它会在内部缓存该 SQL 语句以便日后使用，因此要避免用于最常用操作的 SQL 生成的任何运行时开销。你已经看到了可以如何重写 CRUD 中的 R，因此现在，我们为 CUD 做相同处理。对于每个实体，你都可以使用 Hibernate 注解 @SQLInsert、@SQLUpdate 和 @SQLDelete 分别定义自定义的 CUD SQL 语句。见代码清单 17.3。

代码清单 17.3 用于 User 实体的自定义 DML

路径：/model/src/main/java/org/jpwh/model/customsql/User.java

```
@org.hibernate.annotations.SQLInsert(
    sql = "insert into USERS " +
        "(ACTIVATED, USERNAME, ID) values (?, ?, ?)"
)
@org.hibernate.annotations.SQLUpdate(
    sql = "update USERS set " +
```

```

"ACTIVATED = ?", " +
"USERNAME = ? " +
"where ID = ?"
)
@org.hibernate.annotations.SQLDelete(
    sql = "delete from USERS where ID = ?"
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}

```

需要小心使用用于 SQL 语句和参数占位符(语句中的?)的参数绑定。对于 CUD 自定义, Hibernate 仅支持位置参数。

什么才是参数的正确顺序? Hibernate 将参数绑定到用于 CUD 语句的 SQL 参数有一个内部顺序。弄清楚正确 SQL 语句和参数顺序的最容易方式是, 让 Hibernate 为你生成一个示例。不需要任何自定义的 SQL 语句, 为 `org.hibernate.persister.entity` 目录启用 DEBUG 日志, 并且搜索像下面这样的 Hibernate 启动输出行:

```

Static SQL for entity: org.jpwh.model.customsql.User
Insert 0: insert into USERS (activated, username, id) values (?, ?, ?)
Update 0: update USERS set activated=?, username=? where id=?
Delete 0: delete from USERS where id=?

```

这些自动生成的 SQL 语句显示出了正确的参数顺序, 并且 Hibernate 总是会以该顺序绑定值。将你希望自定义的 SQL 语句复制到注解中, 并且进行必要的修改。

一个特殊的用例是使用 `@SecondaryTable` 映射到另一个表的实体类的属性。到目前为止我们已经显示出的 CUD 自定义语句仅涉及主实体表的列。Hibernate 仍旧会为辅助表中行的插入、更新以及删除自动执行生成的 SQL 语句。

可以通过将 `@org.hibernate.annotations.Table` 注解添加到实体类并且设置其 `sqlInsert`、`sqlUpdate` 和 `sqlDelete` 属性来自定义此 SQL。

如果选择在 XML 中使用 CUD SQL 语句, 那么唯一选择就是在 Hibernate XML 元数据文件中映射整个实体。用于自定义 CUD 语句的这个专有映射格式中的元素是 `<sql-insert>`、`<sql-update>` 和 `<sql-delete>`。幸运的是, CUD 语句通常比查询更为普通, 因此注解适用于大多数应用程序。

现在已经为实体实例的 CRUD 操作添加了自定义 SQL 语句。接下来将介绍如何为加载和修改一个集合重写 SQL 语句。

17.3.3 自定义集合操作

我们来重写加载 `Item#images` 集合时 Hibernate 使用的 SQL 语句。这是一个用 `@ElementCollection` 映射的可嵌入组件集合; 其程序与用于基本类型或多值实体关联 (`@OneToMany` 或 `@ManyToMany`) 集合的程序相同。见代码清单 17.4。

代码清单 17.4 使用一个自定义查询加载集合

路径: /model/src/main/java/org/jpwh/model/customsql/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @org.hibernate.annotations.Loader(namedQuery = "loadImagesForItem")
    protected Set<Image> images = new HashSet<Image>();

    // ...
}
```

像之前一样,你声明了一个会加载该集合的命名查询。不过,这一次,你必须在 Hibernate XML 元数据文件中声明并且映射该查询的结果,这是支持将查询结果映射到集合属性的唯一工具。

路径: /model/src/main/resources/customsql/ItemQueries.hbm.xml

```
<sql-query name="loadImagesForItem">
  <load-collection alias="img" role="Item.images"/>
  select
    ITEM_ID, FILENAME, WIDTH, HEIGHT
  from
    ITEM_IMAGES
  where
    ITEM_ID = ?
</sql-query>
```

该查询必须具有一个(位置或命名)参数。Hibernate 会将它的值设置到拥有该集合的实体标识符。无论何时 Hibernate 需要初始化 Item#images 集合,现在 Hibernate 都会执行自定义 SQL 查询。

有时不必重写用于加载一个集合的整个 SQL 语句:例如,可能你只希望将一个限制添加到所生成的 SQL 语句。我们假设 Category 实体有一个 Item 引用的集合,并且该 Item 有一个激活标记。如果属性 Item#active 为 false,那么你就不会希望在访问 Category#items 集合时加载它。可以在集合映射上使用 Hibernate @Where 注解将这个限制附加到 SQL 语句,作为一个普通的 SQL 片段:

路径: /model/src/main/java/org/jpwh/model/customsql/Category.java

```
@Entity
public class Category {

    @OneToMany(mappedBy = "category")
    @org.hibernate.annotations.Where(clause = "ACTIVE = 'true'")
    protected Set<Item> items = new HashSet<Item>();

    // ...
}
```


也可以写自定义插入和删除集合元素的 SQL 语句，如代码清单 17.5 所示。

代码清单 17.5 用于集合修改的自定义 CUD 语句

路径: /model/src/main/java/org/jpwh/model/customsql/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @org.hibernate.annotations.SQLInsert(
        sql = "insert into ITEM_IMAGES " +
            "(ITEM_ID, FILENAME, HEIGHT, WIDTH) " +
            "values (?, ?, ?, ?)"
    )
    @org.hibernate.annotations.SQLDelete(
        sql = "delete from ITEM_IMAGES " +
            "where ITEM_ID = ? and FILENAME = ? and HEIGHT = ? and WIDTH = ?"
    )
    @org.hibernate.annotations.SQLDeleteAll(
        sql = "delete from ITEM_IMAGES where ITEM_ID = ?"
    )
    protected Set<Image> images = new HashSet<Image>();

    // ...
}
```

要找出正确的参数顺序，可以在添加你的自定义 SQL 注解之前，为 `org.hibernate.persister.collection` 目录启用 DEBUG 日志并且搜索用于这个集合的所生成 SQL 语句的 Hibernate 启动输出。

这里有一个新的注解 `@SQLDeleteAll`，它只应用到基本或可嵌入类型的集合。Hibernate 会在必须从数据库中移除整个集合时执行这个 SQL 语句：例如，在你调用 `someItem.getImages().clear()` 或 `someItem.setImages(new HashSet())` 的时候。

没有必须为这个集合使用的 `@SQLUpdate` 语句，因为 Hibernate 不会为这个可嵌入类型的集合更新行。当一个 Image 属性的值发生变更时，Hibernate 会将其作为集合中的一个新 Image 来检测(回想一下，图片是根据其所有属性值来比较的)。Hibernate 会删除旧的行并且插入一个新行来持久化此变更。

不是延迟加载集合元素，而是在加载所持有实体时急抓取它们。还可以使用一个自定义 SQL 语句重写这个查询。

17.3.4 在自定义加载器中急抓取

我们来思考 `Item#bids` 集合及其加载方式。Hibernate 默认会启用延迟加载，因为你使用 `@OneToMany` 进行了映射，所以仅在你开始遍历集合元素时，Hibernate 才会执行一个查询并且检索数据。因此，在加载 Item 实体时，不必加载任何集合数据。

如果希望在加载该 Item 时急抓取 `Item#bids` 集合，那么首先要在 Item 类上启用一个自定义加载器查询：

路径: /model/src/main/java/org/jpwh/model/customsql/Item.java

```
@org.hibernate.annotations.Loader(
    namedQuery = "findItemByIdFetchBids"
```

```
)
```

```
@Entity
```

```
public class Item {
```

```
    @OneToMany(mappedBy = "item")
```

```
    protected Set<Bid> bids = new HashSet<>();
```

```
    // ...
```

```
}
```

正如上一节一样，必须在 Hibernate XML 元数据文件中声明此命名查询；没有可用的注解用于使用命名查询抓取集合。这里是在单个 OUTER JOIN 中加载一个 Item 及其 bids 集合的 SQL 语句：

路径: /model/src/main/resources/customsql/ItemQueries.hbm.xml

```
<sql-query name="findItemByIdFetchBids">
```

```
    <return alias="i" class="Item"/>
```

```
    <return-join alias="b" property="i.bids"/>
```

```
    select
```

```
        {i.*}, {b.*}
```

```
    from
```

```
        ITEM i
```

```
    left outer join BID b
```

```
        on i.ID = b.ITEM_ID
```

```
    where
```

```
        i.ID = ?
```

```
</sql-query>
```

之前你在本章“急抓取集合”一节的 Java 代码中看到过这个查询和结果映射。这里，你要将一个额外的限制仅应用到 ITEM 的一行，使用指定主键值。

还可以使用一个自定义 SQL 语句急加载像 @ManyToOne 这样的单值实体关联。我们假设你希望在从数据库检索 Bid 实体时急加载 bidder。首先，启用一个命名查询作为实体加载器：

路径: /model/src/main/java/org/jpwh/model/customsql/Bid.java

```
@org.hibernate.annotations.Loader(
    namedQuery = "findBidByIdFetchBidder"
```

```
)
```

```
@Entity
```

```
public class Bid {
```

```
    @ManyToOne(optional = false, fetch = FetchType.LAZY)
```

```
    protected User bidder;
```

```
    // ...
```

```
}
```

不同于用来加载集合的自定义查询，可以使用标准注解声明这个命名查询(当然，还可以在 XML 元数据文件中使用它，用 JPA 或 Hibernate 语法)：

路径：/model/src/main/java/org/jpwh/model/customsql/Bid.java

```
@NamedNativeQueries({
    @NamedNativeQuery(
        name = "findBidByIdFetchBidder",
        query =
            "select " +
            "b.ID as BID_ID, b.AMOUNT, b.ITEM_ID, b.BIDDER_ID, " +
            "u.ID as USER_ID, u.USERNAME, u.ACTIVATED " +
            "from BID b join USERS u on b.BIDDER_ID = u.ID " +
            "where b.ID = ?",
        resultSetMapping = "BidBidderResult"
    )
})
@Entity
public class Bid {
    // ...
}
```

INNER JOIN 适用于这个 SQL 查询，因为 Bid 总是有一个 bidder 并且 BIDDER_ID 外键列绝不会为 NULL。你要在该查询中重命名重复的 ID 列，并且由于你将它们重命名为 BID_ID 和 USER_ID，因此一个自定义结果映射是必要的：

路径：/model/src/main/java/org/jpwh/model/customsql/Bid.java

```
@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "BidBidderResult",
        entities = {
            @EntityResult(
                entityClass = Bid.class,
                fields = {
                    @FieldResult(name = "id", column = "BID_ID"),
                    @FieldResult(name = "amount", column = "AMOUNT"),
                    @FieldResult(name = "item", column = "ITEM_ID"),
                    @FieldResult(name = "bidder", column = "BIDDER_ID")
                }
            ),
            @EntityResult(
                entityClass = User.class,
                fields = {
                    @FieldResult(name = "id", column = "USER_ID"),
                    @FieldResult(name = "username", column = "USERNAME"),
                    @FieldResult(name = "activated", column = "ACTIVATED")
                }
            )
        }
    )
})
```



```
    })
    @Entity
    public class Bid {
        // ...
    }
```

Hibernate 会执行这个自定义 SQL 查询并且在加载 Bid 类的一个实例时，通过 `em.find(Bid.class, BID_ID)` 或者在它必须初始化 Bid 代理时映射结果。Hibernate 会立即加载 Bid#bidder 并且重写关联上的 FetchType.LAZY 设置。

你现在已经用你自己的 SQL 语句自定义了 Hibernate 操作。我们继续处理存储过程并且探究用于将它们集成到 Hibernate 应用程序中的选项。

17.4 调用存储过程

存储过程是数据库应用程序开发中常常用到的。将代码移动得更接近于数据和在数据库内部执行它具有不同的优势。最终不会在每个访问数据的程序中重复功能和逻辑。另外一个观点认为，大量的业务逻辑都不应该重复，因此始终都可以应用它。这包括确保数据完整性的程序：例如，太过复杂而难以声明式实现的约束。你通常还会发现数据中包含程序完整性规则的触发器。

对于所有像报告和统计分析这样的基于大量数据的处理来说，存储过程拥有优势。你应该总是尝试避免将大数据集移动到你的网络上以及数据库和应用程序服务器之间，因此存储过程是海量数据操作自然而然的选择。

当然，肯定有甚至是使用存储过程来实现最基础 CRUD 操作的(遗留)系统。在一个关于这个主题的变化中，有些系统不允许任何对 SQL INSERT、UPDATE 或 DELETE 的直接使用，而是仅能用存储过程调用；这些系统过去也有(并且有时候仍然具有)其存在的价值。

在一些 DBMS 中，除了存储过程，或者说为了替代存储过程，可以声明用户定义的函数。表 17-1 中的汇总显示了过程和函数之间的一些区别。

表 17-1 比较数据库过程和函数

存 储 过 程	函 数
可以使用输入和/或输出参数	可以使用输入参数
返回零、单个或多个值	必须返回一个值(尽管这个值可能不是标量或 NULL)
只能使用 JDBC CallableStatement 来直接调用	可以直接在 CRUD 语句的 SELECT、WHERE 或其他子句中直接调用

除了这些明显区别之外，很难归纳与比较过程和函数的区别。这是 DBMS 支持差异很大的一个地方；有些 DBMS 不支持存储过程或用户定义的函数，而其他的则把这两者混成一体(例如，PostgreSQL 仅有用户定义函数)。存储过程的编程语言通常是专用的。有些数据库甚至支持用 Java 编写存储过程。标准化 Java 存储过程是 SQLJ 项目努力的一个目标，但遗憾的是它没有取得成功。

在这一节中，我们将介绍如何将 Hibernate 与 MySQL 存储过程以及 PostgreSQL 用户定义函数集成使用。首先，我们看看用标准化 Java 持久化 API 和原生 Hibernate API 定义和调用存储过程。然后，我们要用过程调用自定义并且替换 Hibernate CRUD 操作。在阅读这一节之前你要阅读完前面几节，这很重要，因为存储过程的集成依赖于像 Hibernate 中其他 SQL 自定义一样的相同映射选项。

就像本章之前的内容一样，我们在示例中介绍的实际 SQL 存储过程很简单，因此可以专注于更重要的部分——如何在你的应用程序中调用过程并且使用 API。

在调用一个存储过程时，你通常会希望为过程提供输入并且接收其输出。可以在这些过程之间进行区分：

- 返回一个结果集
- 返回多个结果集
- 更新数据并且返回更新行的数量
- 接受输入和/或输出参数
- 返回一个游标，指向数据库中的一个结果

我们首先处理最简单的例子：没有任何参数并且只返回一个结果集中数据的存储过程。

17.4.1 返回一个结果集

可以在 MySQL 中创建以下过程。它会返回一个包含 ITEM 表所有行的结果集：

路径：/model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create procedure FIND_ITEMS()
begin
    select * from ITEM;
end
```

使用一个 EntityManager，构建一个 StoredProcedureQuery 并且执行它：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "FIND_ITEMS",
    Item.class                                     ← 或者结果集映射的名称
);

List<Item> result = query.getResultList();
for (Item item : result) {
    // ...
}
```

就像你之前在本章内容中所看到的一样，Hibernate 会自动将结果集中返回的列映射到 Item 类的属性。由这个查询返回的 Item 实例会被托管并且处于持久化状态。要自定义所返回列的映射，需要提供结果集映射的名称而非 Item.class 参数。

Hibernate 特性

在 Session 上使用 Hibernate 原生 API，你会得到 ProcedureCall 的输出：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("FIND_ITEMS", Item.class);

org.hibernate.result.ResultSetOutput resultSetOutput =
    (org.hibernate.result.ResultSetOutput) call.getOutputs().getCurrent();

List<Item> result = resultSetOutput.getResultList();
```

Hibernate `getCurrent()` 方法已经表明，一个过程可以返回多个 `ResultSet`。一个过程可以返回多个结果集，甚至在它修改了数据时返回更新计数。

17.4.2 返回多个结果以及更新计数

以下 MySQL 过程会返回 ITEM 表还未核准的所有行以及已经核准的所有行，还会为这些行设置 APPROVED 标记：

路径：/model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create procedure APPROVE_ITEMS ()
begin
    select * from ITEM where APPROVED = 0;
    select * from ITEM where APPROVED = 1;
    update ITEM set APPROVED = 1 where APPROVED = 0;
end
```

在该应用程序中，你会得到两个结果集和一个更新计数。访问与处理一个过程调用的结果有一点复杂，但 JPA 与普通 JDBC 是紧密对应的，因此如果已经使用过存储过程，那么应该很熟悉此类代码：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "APPROVE_ITEMS",
    Item.class
);

boolean isCurrentReturnResultSet = query.execute();

while (true) {
    if (isCurrentReturnResultSet) {
        List<Item> result = query.getResultList();
        // ...
    } else {
        int updateCount = query.getUpdateCount();
        if (updateCount > -1) {
            // ...
        }
    }
}
```

②处理结果

③处理结果集

或者结果集映射的名称

①调用 execute()

没有更多的更新计数：跳出循环


```

    } else {
        break;
    }
}

isCurrentReturnResultSet = query.hasMoreResults();

```

④ 处理更新计数

⑤ 前进到下一个结果

- ❶ 用 `execute()` 执行过程调用。这个方法会在调用的首个结果是结果集时返回 `true`，而在首个结果是更新计数时返回 `false`。
- ❷ 在一个循环中处理调用的所有结果。在没有更多结果时停止循环，这总是通过 `hasMoreResults()` 返回 `false` 以及 `getUpdateCount()` 返回 -1 来表示的。
- ❸ 如果当前结果是一个结果集，则读取并处理它。Hibernate 会将每个结果集中的列映射到 `Item` 类的托管实例。或者，提供一个适用于由调用返回的所有结果集的结果集映射名称。
- ❹ 如果当前结果是一个更新计数，则 `getUpdateCount()` 会返回一个大于 -1 的值。
- ❺ `hasMoreResults()` 前进到下一个结果并且指出该结果的类型。

Hibernate 特性

该替代方式——使用 Hibernate API 的过程执行——可能看起来更为直观。它隐藏了验证每个结果类型以及验证是否有更多的过程输出要处理的一些复杂性：

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java`

```

org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("APPROVE_ITEMS", Item.class);

org.hibernate.procedure.ProcedureOutputs callOutputs = call.getOutputs();
org.hibernate.result.Output output;
while ((output = callOutputs.getCurrent()) != null) {
    if (output.isResultSet()) {
        List<Item> result =
            ((org.hibernate.result.ResultSetOutput) output)
                .getResultList();
        // ...
    } else {
        int updateCount =
            ((org.hibernate.result.UpdateCountOutput) output)
                .getUpdateCount();
        // ...
    }
    if (!callOutputs.goToNext())
        break;
}

```

❶ 检查是否有更多输出

❷ 输出是不是一个结果集？

❸ 输出是一个更新计数

❹ 继续

- ❶ 只要 `getCurrent()` 不返回 `null`，就有更多的输出要处理。
- ❷ 输出可以是一个结果集：验证并且转换它。

③ 如果一个输出并非一个结果集，则是一个更新计数。

④ 如果有的话，则继续处理下一个输出。

接下来，我们思考一下具有输入和输出参数的存储过程。

17.4.3 设置输入和输出参数

以下 MySQL 过程会从 ITEM 表返回指定标识符的行，以及商品的总数量：

路径：/model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create procedure FIND_ITEM_TOTAL(in PARAM_ITEM_ID bigint,
                                out PARAM_TOTAL bigint)
begin
    select count(*) into PARAM_TOTAL from ITEM;
    select * from ITEM where ID = PARAM_ITEM_ID;
end
```

下面这个过程会返回一个具有 ITEM 行数据的结果集。此外，设置了其输出参数 PARAM_TOTAL。要在 JPA 中调用这个过程，必须首先注册所有的参数：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "FIND_ITEM_TOTAL",
    Item.class
);
query.registerStoredProcedureParameter(1, Long.class, ParameterMode.IN);
query.registerStoredProcedureParameter(2, Long.class, ParameterMode.OUT);
query.setParameter(1, ITEM_ID);
List<Item> result = query.getResultList();
for (Item item : result) {
    // ...
}
Long totalNumberOfItems = (Long) query.getOutputParameterValue(2);
```

① 注册参数

② 绑定参数值

③ 检索结果

④ 访问参数值

① 根据位置(从 1 开始)及其类型来注册所有的参数。

② 将值绑定到输入参数。

③ 检索由过程返回的结果集。

④ 在你检索了结果集之后，可以访问输出参数值。

还可以注册和使用命名参数，但你不能在某特定调用中混合使用命名和位置参数。另外，还要注意，在 Java 代码中选择的任何参数名称都不必匹配存储过程声明中的参数名称。最终，必须按照在过程签名中声明的相同顺序来注册参数。

Hibernate 特性

原生的 Hibernate API 简化了参数注册和使用：

路径: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```

org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("FIND_ITEM_TOTAL", Item.class);

call.registerParameter(1, Long.class, ParameterMode.IN)
    .bindValue(ITEM_ID);

ParameterRegistration<Long> totalParameter =
    call.registerParameter(2, Long.class, ParameterMode.OUT);

org.hibernate.procedure.ProcedureOutputs callOutputs = call.getOutputs();

org.hibernate.result.Output output;
while ((output = callOutputs.getCurrent()) != null) {
    if (output.isResultSet()) {
        org.hibernate.result.ResultSetOutput resultSetOutput =
            (org.hibernate.result.ResultSetOutput) output;
        List<Item> result = resultSetOutput.getResultList();
        for (Item item : result) {
            // ...
        }
    }
    if (!callOutputs.goToNext())
        break;
}

Long totalNumberOfItems =
    callOutputs.getOutputParameterValue(totalParameter);

```

①注册参数
②得到注册项
③处理结果集
④访问参数值

- ① 注册所有的参数；可以直接绑定输入值。
- ② 稍后可以重用输出参数注册来读取输出值。
- ③ 在访问任何输出参数之前处理所有返回的结果集。
- ④ 通过注册访问输出参数值。

以下 MySQL 过程会使用输入参数和新的商品名称来更新 ITEM 表中的一行：

路径: /model/src/main/resources/querying/StoredProcedures.hbm.xml

```

create procedure UPDATE_ITEM(in PARAM_ITEM_ID bigint,
                             in PARAM_NAME varchar(255))
begin
    update ITEM set NAME = PARAM_NAME where ID = PARAM_ITEM_ID;
end

```

这个过程不会返回任何结果集，因此执行很简单，并且只会得到一个更新计数：

路径: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```

StoredProcedureQuery query = em.createStoredProcedureQuery(
    "UPDATE_ITEM"
);

query.registerStoredProcedureParameter("itemId", Long.class,

```



```

ParameterMode.IN);
query.registerStoredProcedureParameter("name", String.class,
    ParameterMode.IN);
query.setParameter("itemId", ITEM_ID);
query.setParameter("name", "New Item Name");

assertEquals(query.executeUpdate(), 1);    ← 更新计数是 1

// Alternative:
// assertFalse(query.execute());          ← 第一个结果不是结果集
// assertEquals(query.getUpdateCount(), 1);

```

在这个示例中，还可以看到命名参数如何工作以及 Java 代码中的名称不必匹配存储过程声明中的名称。不过，参数注册的顺序仍然很重要；PARAM_ITEM_ID 必须是第一个参数，而 PARAM_ITEM_NAME 必须是第二个。

Hibernate 特性

调用不返回结果集而是修改数据的存储过程的快捷方式是 `executeUpdate()`：其返回值是你的更新计数。或者，可以执行该过程并且调用 `getUpdateCount()`。

这是使用 Hibernate API 的相同过程执行：

路径：/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```

org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("UPDATE_ITEM");

call.registerParameter(1, Long.class, ParameterMode.IN)
    .bindValue(ITEM_ID);

call.registerParameter(2, String.class, ParameterMode.IN)
    .bindValue("New Item Name");

org.hibernate.result.UpdateCountOutput updateCountOutput =
    (org.hibernate.result.UpdateCountOutput) call.getOutputs().getCurrent();

assertEquals(updateCountOutput.getUpdateCount(), 1);

```

由于你知道这个过程不会返回结果集，所以可以直接将首个(当前)输出转换为 `UpdateCountOutput`。

接下来，我们来看看返回一个游标引用而不是结果集的过程。

17.4.4 返回一个游标

MySQL 不支持从存储过程返回游标。以下示例仅适用于 PostgreSQL。这个存储过程(或者，由于这在 PostgreSQL 中是相同的，所以也可以是用户定义的函数)会返回一个指向 ITEM 表所有行的游标：

路径：/model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create function FIND_ITEMS() returns refcursor as $$
```

```

declare someCursor refcursor;
begin
    open someCursor for select * from ITEM;
    return someCursor;
end;
$$ language plpgsql;

```

JPA 总是会使用特殊的 `ParameterMode.REF_CURSOR` 将游标结果注册为参数：

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java`

```

StoredProcedureQuery query = em.createStoredProcedureQuery(
    "FIND_ITEMS",
    Item.class
);

query.registerStoredProcedureParameter(
    1,
    void.class,
    ParameterMode.REF_CURSOR
);

List<Item> result = query.getResultList();
for (Item item : result) {
    // ...
}

```

该参数的类型是 `void`，因为其唯一目的就是在内部准备好用于读取数据的使用游标的调用。当调用 `getResultList()` 时，Hibernate 就知道如何得到期望的输出。

Hibernate 特性

Hibernate API 还提供了游标输出参数的自动处理：

路径: `/examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java`

```

org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("FIND_ITEMS", Item.class);

call.registerParameter(1, void.class, ParameterMode.REF_CURSOR);

org.hibernate.result.ResultSetOutput resultSetOutput =
    (org.hibernate.result.ResultSetOutput) call.getOutputs().getCurrent();

List<Item> result = resultSetOutput.getResultList();
for (Item item : result) {
    // ...
}

```

滚动存储过程游标

在 14.3.3 节中，我们探讨过如何使用一个数据库游标来滚动可能很大的结果集。遗憾的是，在编写本书时，此功能还不能用于 JPA 或 Hibernate API 中的存储过程调用。Hibernate 总是会在内存中检索由存储过程游标引用表示的结果集。

要支持跨 DBMS 方言的数据库游标很难，并且 Hibernate 有一些限制。例如，使用 PostgreSQL，那么游标参数就必须总是第一个注册的参数，并且(由于它是一个函数)数据库只应该返回一个游标。使用 PostgreSQL 方言，如果没有游标返回，那么 Hibernate 就不支持命名参数绑定：你必须使用位置参数绑定。可以查阅你的 Hibernate SQL 方言以了解更多信息；相关的方法有 `Dialect#getResultSet(CallableStatement)` 等。

到此就完成了我们对于直接存储过程调用的 API 的探讨。接下来，还可以使用存储过程来重写由 Hibernate 在其加载或存储数据时生成的语句。

Hibernate 特性

17.5 将存储过程用于 CRUD

你编写的第一个自定义的 CRUD 操作会加载 User 类的一个实体实例。在本章早前的内容中，你使用了一个具有加载器的原生 SQL 查询来实现这一需求。如果必须调用一个存储过程来加载 User 的一个实例，则其过程会同样简单。

17.5.1 自定义一个具有过程的加载器

首先，编写一个调用存储过程的命名查询——例如，在 User 类的注解中：

路径：/model/src/main/java/org/jpwh/model/customsql/procedures/User.java

```
@NamedNativeQueries({
    @NamedNativeQuery(
        name = "findUserById",
        query = "{call FIND_USER_BY_ID(?)}",
        resultClass = User.class
    )
})
@org.hibernate.annotations.Loader(
    namedQuery = "findUserById"
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```

将其与之前 17.3.1 节中的自定义做比较：该加载器的声明仍然是相同的，并且它依赖在任何支持的语言中的已定义命名查询。你仅修改了该命名查询，也可以将它移动到一个 XML 元数据文件中以便将来隔离和分离此关注点。

JPA 未标准化进入 `@NamedNativeQuery` 中的内容，可以随意编写任何 SQL 语句。在大括号中使用 JDBC 转义语法，你就是在表明，“让 JDBC 驱动弄清楚这里要做什么。”如果 JDBC 驱动以及 DBMS 不理解存储过程，那么可以使用 `{call PROCEDURE}` 来调用一个过

程。Hibernate 预期该过程会返回一个结果集，并且该结果集中的第一行预期具有构造一个 User 实例所必须的列。我们之前在 17.3.1 节中列出了所需的列和属性。还要记住，如果过程所返回的列(名称)不太正确或者你不能修改该过程代码，那么你总是可以应用一个结果集映射。

该存储过程必须具有一个匹配带有单个参数的调用的签名。Hibernate 会在加载一个 User 实例时设置这个标识符参数。这里是 MySQL 中具有这样一个签名的存储过程的示例：

路径：/model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure FIND_USER_BY_ID(in PARAM_USER_ID bigint)
begin
    select * from USERS where ID = PARAM_USER_ID;
end
```

接下来，我们要将创建、更新和删除一个 User 映射到一个存储过程。

17.5.2 用于 CUD 的过程

你要使用 Hibernate 注解 @SQLInsert、@SQLUpdate 和 @SQLDelete 来自定义 Hibernate 如何在数据库中创建、更新和删除一个实体实例。可以调用一个存储过程而不是一个自定义 SQL 语句来执行操作：

路径：/model/src/main/java/org/jpwh/model/customsql/procedures/User.java

```
@org.hibernate.annotations.SQLInsert(
    sql = "{call INSERT_USER(?, ?, ?)}",
    callable = true
)
@org.hibernate.annotations.SQLUpdate(
    sql = "{call UPDATE_USER(?, ?, ?)}",
    callable = true,
    check = ResultCheckStyle.NONE
)
@org.hibernate.annotations.SQLDelete(
    sql = "{call DELETE_USER(?)}",
    callable = true
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```

必须指示 Hibernate 使用 JDBC CallableStatement 替代 PreparedStatement 来执行一个操作；因此设置 callable=true 选项。

正如 17.3.2 节中所阐释的，用于过程调用的参数绑定仅适用于位置参数，并且必须以 Hibernate 预期的顺序来声明它们。你的存储过程必须具有一个匹配签名。这里有一些用于 MySQL 在 USERS 表中插入、更新和删除行的过程示例：

路径: /model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure INSERT_USER(in PARAM_ACTIVATED bit,
                             in PARAM_USERNAME varchar(255),
                             in PARAM_ID bigint)
begin
    insert into USERS (ACTIVATED, USERNAME, ID)
        values (PARAM_ACTIVATED, PARAM_USERNAME, PARAM_ID);
end
```

路径: /model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure UPDATE_USER(in PARAM_ACTIVATED bit,
                             in PARAM_USERNAME varchar(255),
                             in PARAM_ID bigint)
begin
    update USERS set
        ACTIVATED = PARAM_ACTIVATED,
        USERNAME = PARAM_USERNAME
    where ID = PARAM_ID;
end
```

路径: /model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure DELETE_USER(in PARAM_ID bigint)
begin
    delete from USERS where ID = PARAM_ID;
end
```

当一个存储过程插入、更新或删除一个 User 实例时, Hibernate 必须知道该调用是否成功。通常, 对于动态生成的 SQL 来说, Hibernate 会查看从一个操作中返回的更新行的数量。如果启用版本控制(参阅 11.2.2 节), 并且该操作没有或者不能更新任何行, 则会出现乐观锁故障。如果编写你自己的 SQL, 那么也可以自定义这一行为。在更新或删除行时, 针对数据库状态执行版本检查取决于存储过程。在你的注解中使用 `check` 选项, 就能让 Hibernate 知道该过程会如何实现这一需求。

默认设置是 `ResultCheckStyle.NONE`, 并且以下设置是可用的:

- **NONE**——该过程会在操作失败时抛出一个异常。Hibernate 不会执行任何显式检查, 而是依赖过程代码来做正确的事情。如果启用版本控制, 那么你的过程就必须比较/递增版本并且在它检测到版本不匹配时抛出一个异常。
- **COUNT**——过程会执行任何需要的版本递增并且会对更新行的数量进行检查以及将其返回到 Hibernate 作为一个更新计数。Hibernate 会使用 `CallableStatement#updateCount()` 来访问该结果。
- **PARAM**——过程会执行任何需要的版本递增并且会对更新行的数量进行检查以及在其首个输出参数中将其返回到 Hibernate。对于这一检查方式, 需要将一个额外的问号添加到你的调用, 并且在你的存储过程中, 在该(第一个)输出参数中返回 DML 操作的行计数。Hibernate 会自动注册该参数并且在调用完成时读取它的值。

ResultCheckStyle 选项的可用性

在编写本书时, Hibernate 仅实现了 ResultCheckStyle.NONE。

最后, 要记住, 有时不能在 Hibernate 中映射存储过程和函数。在这种情况下, 必须回退到普通的 JDBC。有时可以用另一个具有 Hibernate 预期参数接口的存储过程来包装一个遗留的存储过程。

17.6 本章小结

- 介绍了如何在必要时回退到 JDBC API。即便是对于自定义 SQL 查询, Hibernate 也会承担繁重的任务并且使用灵活的映射选项将 ResultSet 转换成域模型类的实例, 其中包括自定义结果映射。还可以外部化原生查询, 以便得到一个更干净的设置。
- 探讨了如何重写和提供你自己的用于常规创建、读取、更新和删除(CRUD)操作以及集合操作的 SQL 语句。
- 可以启用自定义加载器并且在这样的加载器中使用急抓取。
- 学习了如何直接调用数据库存储过程以及将它们集成到 Hibernate 中。探究了单结果集的处理以及操作结果集和更新计数。设置了存储过程(输入和输出)参数并且学习了如何返回一个数据库游标。还介绍了如何将存储过程用于 CRUD。

第 V 部分

构建应用程序

在本书第 V 部分中，我们将探讨分层以及具有会话意识的 Java 数据库应用程序的设计和实现。我们将探讨与 Hibernate 一同使用的最常用设计模式，比如数据访问对象 (DAO)。你将看到如何才能轻松测试你的 Hibernate 应用程序，以及一般而言，如果处理 Web 中的 ORM 软件和客户端/服务器应用程序，有哪些相关的其他最佳实践。

第 18 章都是关于设计客户端/服务器应用程序的内容。你将学习用于客户端/服务器架构的模式，编写与测试一个持久化层，以及集成 EJB 与 JPA。在第 19 章将查看 Web 应用程序的构建以及将 JPA 与 CDI 和 JSF 集成。我们将介绍表中数据的浏览、长期运行的会话的实现，以及实体序列化的自定义。最后，在第 20 章中，我们将查看通过执行大量和批量数据操作来扩展 Hibernate，并且使用共享缓存来改进可扩展性。

在阅读完本部分之后，你将具有架构级别的知识，以便构建一个完整应用程序并且让它能随着其成功而扩展规模。

设计客户端/服务器

第 18 章

应用程序

18

本章内容简介：

- 用于客户端/服务器架构的模式
- 编写和测试一个持久化层
- 集成 EJB 与 JPA

大多数 JPA 开发人员都会通过 Hibernate 访问数据库层的基于 Java 的服务器来构建客户端/服务器应用程序。了解 EntityManager 和系统事务如何工作之后，你大概就能找到你自己的服务器架构了。你必须弄明白在何处创建 EntityManager、何时以及如何关闭它，还有如何设置事务边界。

你可能会思考，从客户端发起请求到客户端得到响应与服务器上的持久化上下文和事务之间是什么关系。单个系统事务应该处理每一个客户端请求吗？几个连续请求可以保持一个持久化上下文的打开状态吗？分离的实体状态如何适用于此场景？可以并且应该序列化客户端和服务端之间的实体数据吗？这些决策会如何影响你的客户端设计？

在开始回答这些问题之前，必须说一下，我们不会在本章探讨除了 JPA 和 EJB 之外的任何特殊框架。除了 JPA 外，代码示例只使用 EJB 有几个原因：

- 我们的目标是专注于使用 JPA 的客户端/服务器设计模式。像客户端和服务端之间的数据序列化这样的许多交叉问题，都在 EJB 中标准化了，因此我们不必立即解决每一个问题。我们知道你可能不会编写一个 EJB 客户端应用程序。不过，使用本章中的示例 EJB 客户端代码，你将具有明智的决策基础，以便在选择和使用另一种框架时做出你自己的判断。我们将在下一章中探讨自定义序列化过程并且解释如何与任何客户端交换你的 JPA 托管数据。
- 我们不可能介绍 Java 领域中的每一个客户端/服务器框架组合。注意，我们没有将我们的范围限定在 Web 服务器应用程序。当然，Web 应用程序很重要，因此我们将在下一章会专注介绍 JPA 与 JSF 和 JAX-RS。在这一章中，我们主要关注依赖 JPA 进行持久化的任何客户端/服务器系统，以及像 DAO 模式这样的抽象，无论你使用何种框架，它都很有用。

- 即便你仅仅在服务器端使用 EJB，它们也是有效的。它们提供了事务管理，并且可以将持久化上下文绑定到状态性会话 bean。我们还将探讨这些详细细节，因而如果应用程序架构需要服务器端的 EJB，你也会知道如何构建它们。

在本章中，将具有简单明了工作流的两个简单用例实现为一个实际运行的应用程序：编辑一个拍卖商品，并且为一个商品设置出价。首先看看持久化层以及如何才能将 JPA 操作封装到可重用组件中：实际上是使用 DAO 模式。这会提供一个坚实的基础，以便构建更多的应用程序功能。

然后将用例实现为会话：从应用程序用户角度出发的工作单元。你会看到用于无状态和状态性服务器端组件的代码，以及这对于客户端设计和整体应用程序架构的影响。这不仅会影响应用程序的行为，还会影响其可扩展性以及健壮性。我们会用不同的策略重复所有这些示例并且重点介绍其区别。

我们首先介绍如何具体化持久化层和 DAO 模式。

18.1 创建持久化层

在 3.1.1 节中，我们介绍过在一个独立层中持久化代码的封装。尽管 JPA 已经提供了某种级别的抽象，但还有一些原因你应该考虑将 JPA 调用隐藏在表面之后：

- 自定义持久化层可以为数据访问操作提供较高级别的抽象。不公开 `EntityManager` 公开的基础 CRUD 和查询操作，而公开较高级别的操作，比如 `getMaximumBid (Item i)` 和 `findItems (User soldBy)` 方法。这一抽象是在较大应用程序中创建一个持久化层的主要原因：为了支持相同数据访问操作的重用。
- 持久化层可以具有不公开实现详情的通用接口。换句话说，可以对持久化层的任何客户端隐藏你正在使用 `Hibernate` (或 `Java` 持久化) 实现数据访问操作这一事实。我们认为持久化层的移植性是一个不重要的关注点，因为像 `Hibernate` 这样的完整对象/关系映射解决方案已经提供了数据库移植性。你未来不太可能会用不同的软件重写你的持久化层，并且仍旧不会希望修改任何客户端代码。此外，`Java` 持久化是一个标准化并且完全可移植的 API；偶尔将其公开给持久化层的客户端没有什么危害。

持久化层可以统一数据访问操作。这个问题与可移植性有关，不过这是从一个不同的角度来看的。想象一下，你必须处理混合的数据访问代码，比如 JPA 和 JDBC 操作。通过统一客户端看到和使用的表层，可以对客户端隐藏这一实现详情。如果必须处理不同类型的数据存储，那么这就是一个编写持久化层的正当理由。

如果考虑将可移植性和统一性作为创建持久化层的意外效果，那么你的主要动机就是实现较高级别的抽象并且提高可维护性以及数据访问代码的重用。这些都是合理的原因，并且我们支持你在全部但是最简单的应用程序中创建一个具有一个通用表层的持久化层。但应该总是首先考虑直接使用 JPA，而不使用任何额外的分层。保持它尽可能简单，并且在你意识到正在重复相同查询和持久化操作时，在 JPA 上创建一个简洁的持久化层。

许多可用的工具都宣称 JPA 或 `Hibernate` 简化了持久化层的创建。我们建议首先尝

试不使用这样的工具，并且仅在需要一个特定功能时才购买一个产品。要特别当心代码和查询生成器：经常会听到宣称可应对每一个问题的全面解决方案，从长期来看，这种解决方案都会变成一个明显的限制和巨大的维护负担。如果开发过程依赖代码生成工具的运行，那么也会对生产效率造成巨大影响。对于 Hibernate 自己的工具来说，自然也是如此：例如，如果必须在每次变更时都从 SQL 架构生成实体类的源，就会如此。持久化层是应用程序的一个重要部分，并且你必须注意你通过引入额外依赖项所带来的义务。在本章和下一章中，你将看到在不使用任何额外工具的情况下，如何避免通常与持久化层组件相关的重复代码。

设计一个持久化层外观的方式有很多种——有些小的应用程序使用单一 `DataAccess` 类；其他的会将数据访问操作混合到域类中(活动记录模式，本书没有探讨过它)——但我们选用了 DAO 模式。

18.1.1 一种通用的数据访问对象模式

DAO 设计模式源自 15 年前 Sun 的 Java 设计蓝图；它有很长的历史。DAO 类会将一个接口定义到与特定实体相关的持久化操作；它建议你将与该实体持久化有关的代码分组在一起。由于其使用了这么多年，因而该 DAO 模式存在许多变体。图 18-1 中显示了我们推荐的设计的基本结构。

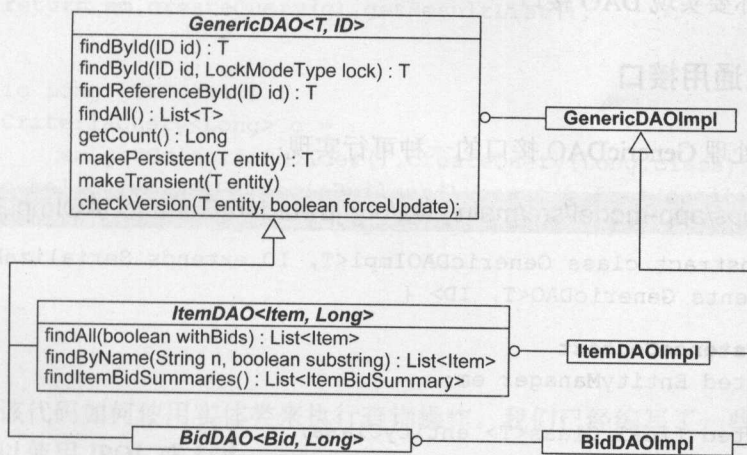


图 18-1 通用的 DAO 接口支持任意实现

我们将该持久化层设计为具有两个平行的层次结构：一侧是接口，另一侧是实现。基本的实例存储和实例检索操作被分组到一个通用的超接口和一个使用特定持久化解决方案(自然是使用 Hibernate)实现这些操作的超类。该通用接口是通过用于特定实体的接口来扩展的，这些实体需要额外的与业务有关的数据访问操作。同样，可以使用实体 DAO 接口的一个或多个实现。

我们快速查看此图中显示的一些接口与方法。其中有大批查找器方法。这些方法通常会返回托管的(处于持久化状态的)实体实例，但它们也可以返回任意数据传输对象，比如 `ItemBidSummary`。查找器方法是最大的代码重复问题；如果不仔细规划，你最终可能会有大量重复。第一个步骤是尝试让它们尽可能通用并且在层次结构中将它们上移，最好是移动到顶层接口中。思考 `ItemDAO` 中的 `findByName()` 方法：你很可能必须马上为商品搜索

添加更多选项，或者你可能希望得到数据库预先分类的结果，或者你可能要实现某种分页功能。稍后我们在 19.2 节中将再次详尽地探讨这一点并且向你介绍用于分类和分页的一种通用解决方案。

由 DAO API 提供的方法清晰地表明，这是一个状态管理持久化层。像 `makePersistent()` 和 `makeTransient()` 这样的方法会修改一个实体实例的状态(或者启用级联一次性修改许多实例的状态)。在修改一个实体实例时，客户端可以预期，持久化引擎会(用刷新)自动执行更新(没有使用 `performUpdate()` 方法)。如果持久化层是面向语句的，那么你就要编写一个完全不同的 DAO 接口：例如，如果没有使用 `Hibernate` 而是仅仅使用普通 `JDBC` 来实现它的话。

我们在这里介绍的持久化层表层不会公开任何 `Hibernate` 或 `Java` 持久化接口给客户端，因此从理论上讲，可以用任何软件而无须修改客户端代码来实现它。你可能不想要或者不需要持久化层的可移植性，就像之前阐释的那样。在那种情况下，你应该考虑公开 `Hibernate` 或 `Java` 持久化接口——例如，可以允许客户端访问 `JPA CriteriaBuilder`，然后使用一个通用 `findBy(CriteriaQuery)` 方法。这个决定取决于你；你可能会认为，公开 `Java` 持久化接口是比公开 `Hibernate` 接口更安全的选择。不过，你应该知道，尽管能够将持久化层的实现从一个 `JPA` 提供程序变更到另一个，但使用普通 `JDBC` 语句重写一个面向状态的持久化层几乎是不可能的。

接下来，你要实现 DAO 接口。

18.1.2 实现通用接口

我们继续处理 `GenericDAO` 接口的一种可行实现：

路径：/apps/app-model/src/main/java/org/jpwh/dao/GenericDAOImpl.java

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    @PersistenceContext
    protected EntityManager em;

    protected final Class<T> entityClass;

    protected GenericDAOImpl(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    // ...
}
```

这个通用实现需要两样东西来运行：`EntityManager` 和实体类。子类必须提供该实体类作为一个构造函数参数。不过，可以通过理解 `@PersistenceContext` 注入注解(比如，任何标准 `Java EE` 容器)的运行时容器或者通过 `setEntityManager()` 来提供 `EntityManager`。

接下来，我们看看查找器方法：

路径: /apps/app-model/src/main/java/org/jpwh/dao/GenericDAOImpl.java

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    // ...

    public T findById(ID id) {
        return findById(id, LockModeType.NONE);
    }

    public T findById(ID id, LockModeType lockModeType) {
        return em.find(entityClass, id, lockModeType);
    }

    public T findReferenceById(ID id) {
        return em.getReference(entityClass, id);
    }

    public List<T> findAll() {
        CriteriaQuery<T> c =
            em.getCriteriaBuilder().createQuery(entityClass);
        c.select(c.from(entityClass));
        return em.createQuery(c).getResultList();
    }

    public Long getCount() {
        CriteriaQuery<Long> c =
            em.getCriteriaBuilder().createQuery(Long.class);
        c.select(em.getCriteriaBuilder().count(c.from(entityClass)));
        return em.createQuery(c).getSingleResult();
    }

    // ...
}
```

可以看到该代码如何使用实体类来执行查询操作。我们已经编写了一些简单的条件查询，但你也可以使用 JPQL 或 SQL。

最后，这里是状态管理操作：

路径: /apps/app-model/src/main/java/org/jpwh/dao/GenericDAOImpl.java

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    // ...

    public T makePersistent(T instance) {
        // merge() handles transient AND detached instances
        return em.merge(instance);
    }

    public void makeTransient(T instance) {
```



```

        em.remove(instance);
    }
    public void checkVersion(T entity, boolean forceUpdate) {
        em.lock(
            entity,
            forceUpdate
            ? LockModeType.OPTIMISTIC_FORCE_INCREMENT
            : LockModeType.OPTIMISTIC
        );
    }
}

```

一个重要的决策是,你如何实现 `makePersistent()` 方法。这里我们选择了 `EntityManager#merge()`, 因为它是最通用的。如果指定参数是一个瞬时实体实例, 那么进行合并将会返回一个持久化实例。如果该参数是一个分离实体实例, 那么进行合并也会返回一个持久化实例。这就为客户端提供了一个一致的 API, 而无须担心调用 `makePersistent()` 之前实体实例的状态。但客户端需要清楚, `makePersistent()` 的返回值总是当前实例, 并且它指定的参数现在必须被丢弃(参阅 10.3.4 节)。

现在已经完成了构建持久化层的基本运行组成以及它公开给系统较上层的通用接口。下一步, 要创建与实体有关的 DAO 接口以及通过扩展通用接口和实现的实现。

18.1.3 实现实体 DAO

到目前为止, 你所创建的所有一切都是抽象和通用的——你甚至不能实例化 `GenericDAOImpl`。现在要通过使用一个具体类来扩展 `GenericDAOImpl` 从而实现该 `ItemDAO` 接口。

首先必须做出与调用方如何访问 DAO 有关的选择。还需要考虑 DAO 实例的生命周期。使用当前设计, 那么除了 `EntityManager` 成员之外, DAO 类就是无状态的。

调用方线程可以共享一个 DAO 实例。例如, 在多线程的 Java EE 环境中, 自动注入的 `EntityManager` 实际上是线程安全的, 因为它通常是在内部作为一个代理来实现的, 这个代理要委托给一些绑定线程或绑定事务的持久化上下文。当然, 如果调用 DAO 上的 `setEntityManager()`, 那么该实例就无法被共享, 并且应该仅由一个(比如集成/单元测试)线程使用。

EJB 无状态会话 bean 池会是一个好的选择, 并且如果将具体的 `ItemDAOImpl` 注解为无状态 EJB 组件的话, 那么线程安全的持久化上下文注入就是可用的:

路径: `/apps/app-model/src/main/java/org/jpwh/dao/ItemDAOImpl.java`

```

@Stateless
public class ItemDAOImpl extends GenericDAOImpl<Item, Long>
    implements ItemDAO {
    public ItemDAOImpl() {
        super(Item.class);
    }
    // ...
}

```

稍后你会看到 EJB 容器如何为注入选择“合适的”持久化上下文。

注入的 EntityManager 的线程安全

Java EE 规范未明确规定使用 `@PersistenceContext` 的已注入 `EntityManager` 的线程安全。JPA 规范规定，“只能以单线程方式访问” `EntityManager`。这就表明，它不能被注入到固有的多线程组件，比如 EJB、单例 beans，以及由于其默认设置而无法运行 `SingleThreadModel` 的 servlet。但 EJB 规范要求，EJB 容器要序列化对每个状态性和无状态会话 bean 实例的调用。因此在无状态或状态性 EJB 中注入的 `EntityManager` 是线程安全的；容器通过注入一个 `EntityManager` 占位符来实现这一点。此外，应用程序服务器可能(并非必然如此)支持对单例 bean 或多线程 servlet 中已注入的 `EntityManager` 进行线程安全的访问。如果对此有疑问，可以注入线程安全的 `EntityManagerFactory`，然后在你的组件服务方法中创建并关闭你自己的应用程序托管的 `EntityManager`。

下面是在 `ItemDAO` 中定义的查找器方法：

路径：/apps/app-model/src/main/java/org/jpwh/dao/ItemDAOImpl.java

@Stateless

```
public class ItemDAOImpl extends GenericDAOImpl<Item, Long>
    implements ItemDAO {
```

```
// ...
```

@Override

```
public List<Item> findAll(boolean withBids) {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
    // ...
    return em.createQuery(criteria).getResultList();
}
```

@Override

```
public List<Item> findByName(String name, boolean substring) {
    // ...
}
```

@Override

```
public List<ItemBidSummary> findItemBidSummaries() {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<ItemBidSummary> criteria =
        cb.createQuery(ItemBidSummary.class);
    // ...
    return em.createQuery(criteria).getResultList();
}
```

在阅读完前面的章节之后，对于编写这些查询你应该没有任何问题了；它们非常简单：要么使用条件查询 API，要么根据名称调用外部化的 JPQL 查询。应该思考用于条件查询的静态元模型。

完成 ItemDAO 之后，就可以继续处理 BidDAO 了：

路径：/apps/app-model/src/main/java/org/jpwh/dao/BidDAOImpl.java

@Stateless

```
public class BidDAOImpl extends GenericDAOImpl<Bid, Long>
    implements BidDAO {

    public BidDAOImpl() {
        super(Bid.class);
    }
}
```

如你所见，这是一个空的 DAO 实现，它只继承了通用方法。在下一节中，我们要探讨你有可能放入这个 DAO 类中的一些操作。我们还没有介绍任何 UserDAO 或 CategoryDAO 代码，假设你要根据需要编写这些 DAO 接口和实现。

我们的下一个主题是测试这个持久化层：应该进行测试吗？如果应该，那么如何进行测试呢？

18.1.4 测试持久化层

到目前为止，本书中的几乎所有示例都是直接从实际的测试代码中提取出来的。在后面所有的示例中，我们还将继续这样做，但我们必须问一问：你应该为持久化层编写测试以验证其功能吗？

根据我们的经验，单独测试持久化层通常没什么意义。可以实例化域 DAO 类并且提供一个虚拟 EntityManager。这样的单元测试具有受限值并且需要大量的编写工作。相反，我们建议你创建集成测试，它会测试较大部分的应用程序栈并且包含数据库系统。本章中的其余所有示例都来自这样的集成测试；它们会模拟一个调用服务器应用程序的客户端，后端使用一个真实的数据库。因此，你是在测试重要的内容：服务的正确行为、域模型依赖的业务逻辑，以及通过 DAO 的数据库访问，全部一起测试。

随之而来的问题就是如何准备这样的集成测试。你希望在真实 Java EE 环境中实际的运行时容器内进行测试。谓词，我们使用 Arquillian(<http://arquillian.org>)，一个集成了 TestNG 的工具。使用 Arquillian，可以在你的测试代码中准备一个虚拟归档处，然后在真实的应用程序服务器上执行它。看看这些示例以便弄明白它如何工作。

一个更加有趣的问题是，为集成测试准备测试数据。大多数有意义的测试都需要一些存在于数据库中的数据。你希望在测试运行之前将那些测试数据加载到数据库中，并且每个测试都应该使用一个干净并且良好定义的数据集，这样就能编写可靠的断言。

根据我们的经验，这里有三种导入测试数据的常用技术：

- 测试固件会在每个测试前执行一个方法来获得一个 EntityManager。要手动实例化测试数据实体并且用 EntityManager API 持久化它们。这一策略的主要优势在于，你将不少映射作为了附带影响来测试。另一个优势是可以轻松使用编程方式来访问测试数据。例如，如果测试代码中需要特定测试 Item 的标识符值，那么它已经在 Java 中了，因为可以从数据导入方法中将它传递回来。其劣势在于，测试数据

难以维护, 因为 Java 代码并非出色的数据格式。可以通过使用 Hibernate 的架构导出功能在每次测试之后丢弃和重建该架构来从数据库中清除测试数据。到目前为止, 本书中的所有集成测试都使用了这种方法; 可以在示例代码中的每个测试的旁边找到测试数据导入程序。

- Arquillian 可以在每次测试运行之前执行一个 DbUnit(<http://dbunit.sourceforge.net>) 数据集导入。DbUnit 为编写数据集提供了几种格式, 其中包括经常使用的平滑 XML 语法。这并非最紧凑的格式, 但容易阅读和维护。本章中的示例使用了这一方法。可以在测试类上找到 Arquillian 的 @UsingDataSet, 它带有要导入的 XML 文件的路径。Hibernate 会生成和丢弃 SQL 架构, 而有了 DbUnit 的帮助, Arquillian 会将测试数据加载到数据库中。如果希望保持测试数据独立于测试之外, 那么这可能就是适合你的方法。如果不使用 Arquillian, 那么使用 DbUnit 手动导入数据集也很简单——参见本章示例中的 SampleDataImporter。我们在开发期间运行示例应用程序时部署了这个导入器, 以便将相同数据用于交互使用, 就像在自动化测试中一样。
- 在 9.1.1 节中, 你看到了如何在 Hibernate 启动时执行自定义 SQL 脚本。加载脚本会在 Hibernate 生成架构之后执行; 这是使用普通 INSERT SQL 语句导入测试数据的一个非常棒的工具。下一章中的示例使用了这一方法。其主要优势在于, 可以将 INSERT 语句从 SQL 控制台复制/粘贴到你的测试固件, 反之亦然。此外, 如果数据库支持 SQL 行值构造函数语法, 那么你就可以编写紧凑的多行插入语句, 比如 insert into MY_TABLE (MY_COLUMN) values (1), (2), (3), ...。

我们将选取一种策略的权力留给你。这通常是一个喜好问题, 也与你必须维护的测试数据量有关。注意, 我们正在谈论用于集成测试的测试数据, 而非性能或扩展性测试。如果需要大量的(大多是随机的)测试数据用于负载测试, 可以考虑使用像 Benerator(<http://databene.org/databene-benerator.html>)这样的数据生成工具。

到此就完成了持久化层的首次迭代。现在可以得到 ItemDAO 实例并且在访问数据库时处理较高级别的抽象。我们来编写一个调用此持久化层的客户端并且实现该应用程序的其余部分。

18.2 构建一个无状态服务器

该应用程序将是一个无状态服务器应用程序, 这意味着服务器上不会在客户端请求之间托管应用程序状态。这个应用程序很简单, 其中它只支持两个用例: 编辑一个拍卖商品以及对商品出价。

将这些工作流视作会话: 从应用程序用户角度来看的工作单元。应用程序用户的观点不一定与我们作为开发人员对于系统的观点相同; 开发人员通常认为系统事务是一个工作单元。现在我们重点看看这一不匹配, 以及用户的观点会如何影响服务器和客户端代码的设计。我们首先处理第一个会话: 编辑一个商品。

18.2.1 编辑一个拍卖商品

客户端是一个基于文本的普通 EJB 控制台应用程序。看看图 18-2 中使用这个客户端进行的“编辑一个拍卖商品”的用户会话。

该客户端为用户呈现了一组拍卖商品；用户要选取一个。然后客户端会询问用户想要执行哪个操作。最后，在输入一个新的名称之后，客户端会显示一条成功确认消息。现在系统准备好下一次会话了。该示例客户端会再次启动并且呈现一组拍卖商品。

图 18-3 中显示了此会话的调用序列。这是本节其余内容的路线图。

```
>>> Starting dialog, connecting to server (press CTRL+C to exit)...

ID | Name | Auction End | Highest Bid
---|---|---|---
1 | Baseball Glove | 06. Mar 2015 15:00 | 13.00
2 | Aquarium | 07. Mar 2015 16:00 | -
3 | Golf GTI | 08. Mar 2015 09:30 | 30000.00
4 | Blade Runner Bluray | 09. Mar 2015 10:20 | -
5 | Coffee Machine | 10. Mar 2015 14:55 | 6.00

Please enter an item ID:
1
Would you like to rename (n) the item or place a bid (b):
n
New name for item 'Baseball Glove':
Pretty Baseball Glove
=> Item name changed successfully!
```

图 18-2 从用户角度看，这个会话是一个工作单元

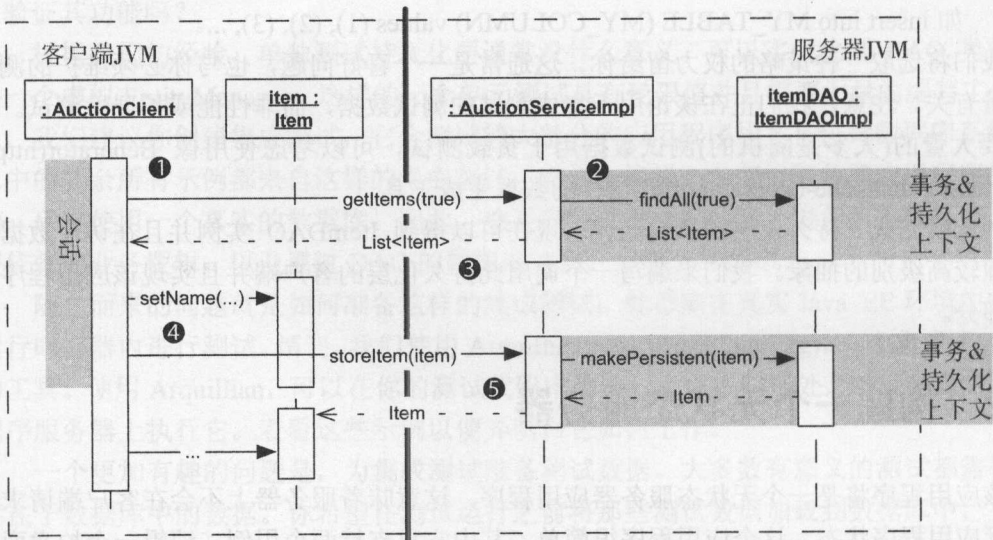


图 18-3 “编辑一个拍卖商品”会话中的调用

我们在代码中更为详细地看看这些内容；可以参考该图中的箭头项来追踪我们的位置。接下来你看到的代码来自一个模拟客户端的测试用例，接着是处理这些客户端调用的服务器端组件的代码。

该客户端会从服务器检索一组 Item 实例来开启会话①并且还会用 true 来请求急抓取 Item#bids 集合。由于服务器未持有会话状态，因此客户端必须完成此任务：

路径: /apps/app-stateless-server/src/test/java/org/jpwh/test/stateless/
AuctionServiceTest.java

```
List<Item> items;           ← 客户端必须管理应用程序状态: 一组商品。
items = service.getItems(true); ← 得到处于分离状态的所有商品, 并且加载出价。
```

服务器端代码会借助 DAO 处理该调用:

路径: /apps/app-stateless-server/src/main/java/org/jpwh/stateless/
AuctionServiceImpl.java

```
@javax.ejb.Stateless
@javax.ejb.Local(AuctionService.class)
@javax.ejb.Remote(RemoteAuctionService.class)
public class AuctionServiceImpl implements AuctionService {

    @Inject
    protected ItemDAO itemDAO;

    @Inject
    protected BidDAO bidDAO;

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED) ← 默认的
    public List<Item> getItems(boolean withBids) {
        return itemDAO.findAll(withBids);
    }

    // ...
}
```

(可以忽略此处声明的接口; 它们很简单, 但对于 EJB 的远程调用和本地测试来说是必要的)。由于在客户端调用 `getItems()` 时没有活动的事务, 所以会开启一个新的事务。该事务会在方法返回时自动提交。在这种情况下 `@TransactionAttribute` 注解是可选的; 默认的行为需要在 EJB 方法调用上使用一个事务。

`getItems()` EJB 方法会调用 `ItemDAO` 来检索 `Item` 实例的一个 `List`②。Java EE 容器会自动查找并且注入 `ItemDAO`, 而 `EntityManager` 会被设置在 `DAO` 上。由于没有 `EntityManager` 或持久化上下文是与当前事务相关联的, 所以会开启一个新的持久化上下文并且将它联结到该事务。事务提交时会刷新和关闭该持久化上下文。这是无状态 EJB 的一个便利功能; 你不必为了在事务中使用 JPA 而做太多工作。

处于分离状态的 `Item` 实例的 `List`(在持久化上下文被关闭之后)会被返回到客户端③。目前你不必担心序列化; 只要 `List` 和 `Item` 以及所有其他可得到的类型是 `Serializable`, 那么 EJB 框架就会负责处理它。

接下来, 客户端会设置所选择的 `Item` 的新名称, 并且通过发送已分离和已修改的 `Item` 来要求服务器存储该变更④:

路径: /apps/app-stateless-server/src/test/java/org/jpwh/test/stateless/
AuctionServiceTest.java

```
detachedItem.setName("Pretty Baseball Glove");
```

```
detachedItem = service.storeItem(detachedItem);
```

调用服务并且让变更持久化。返回当前 Item 实例。

服务器会使用该分离的 Item 实例并且要求 ItemDAO 让变更持久化⑤, 在内部合并修改:

路径: /apps/app-stateless-server/src/main/java/org/jpwh/stateless/
AuctionServiceImpl.java

```
public class AuctionServiceImpl implements AuctionService {
```

```
// ...
```

```
@Override
```

```
public Item storeItem(Item item) {
```

```
    return itemDAO.makePersistent(item);
```

```
}
```

```
// ...
```

```
}
```

更新后的状态——合并的结果——会被返回到客户端。

会话已完成, 并且客户端可能会忽略所返回的已更新 Item。但客户端知道此返回值是最新状态并且任何它在会话期间持有的之前的状态, 比如 Item 实例的 List, 已经过期且应该可能已经被丢弃。随后的会话应该以该最新状态开始: 使用最新返回的 Item, 或通过获得一个新的列表。

现在已经看到了如何实现单个会话——整个工作单元, 从用户角度看——使用服务器上的两个系统事务。因为只会加载第一个系统事务中的数据并且延迟编写变更到最后的事务, 该对话是原子的: 在最后一步完成之后才会持久化变更。我们使用第二个用例来扩展它: 为每个商品放置一个出价。

18.2.2 放置出价

在控制台客户端中, 一个用户的“放置一个出价”的会话看起来会像图 18-4 一样。该客户端会再次为用户呈现一系列拍卖商品并且要求用户选取其中一个。用户可以放置一个出价, 并且如果该出价被成功存储, 则该用户会收到一条成功的确认消息。图 18-5 中显示了调用的序列以及代码路线图。

```
>>> Starting dialog, connecting to server (press CTRL+C to exit)...

ID | Name                | Auction End          | Highest Bid
---|---|---|---
1  | Baseball Glove       | 06. Mar 2015 15:00   | 13.00
2  | Aquarium             | 07. Mar 2015 16:00   | -
3  | Golf GTI             | 08. Mar 2015 09:30   | 30000.00
4  | Blade Runner Bluray  | 09. Mar 2015 10:20   | -
5  | Coffee Machine       | 10. Mar 2015 14:55   | 6.00

Please enter an item ID:
1
Would you like to rename (n) the item or place a bid (b):
b
Your bid for item 'Baseball Glove':
15
=> Bid placed successfully!
```

图 18-4 用户放置出价：从用户角度看的一个工作单元

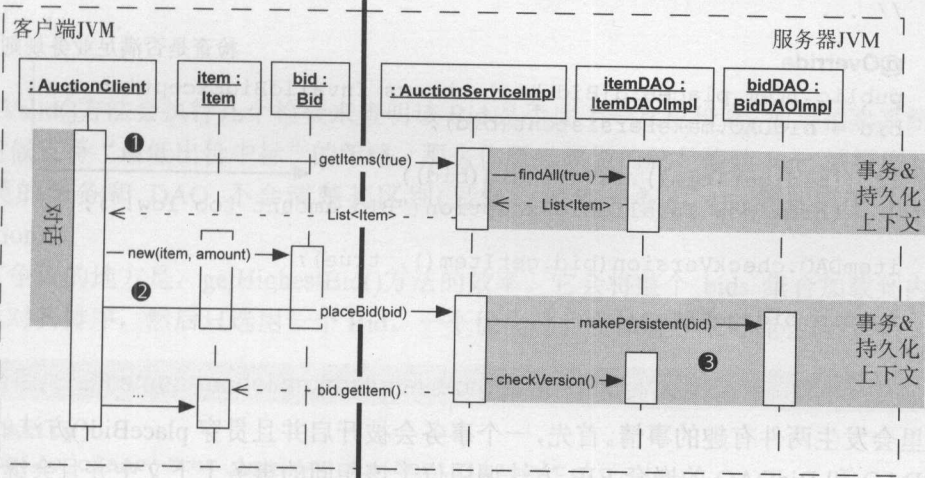


图 18-5 “放置一个出价”会话中的调用

我们再次按步骤处理测试客户端和服务端代码。首先❶，客户端得到一组 Item 实例并且急抓取 Item#bids 集合。你在上一节中看到过用于此目的的代码。

然后，客户端会在接收到用户输入的金额❷之后创建一个新的 Bid 实例，使用分离的已选 Item 链接这个新的瞬时 Bid。客户端必须存储此新 Bid 并且将它发送到服务器。如果未正确记录你的服务 API，那么客户端可能会尝试发送已分离的 Item：

```
路径: /apps/app-stateless-server/src/test/java/org/jpwh/test/stateless/
AuctionServiceTest.java
```

```
item.getBids().add(newBid);
item = service.storeItem(item);
```

“可能这个服务已经在 Item#bids @OneToMany 上级联持久化了？”之后，什么都不会发生——未存储出价。

此处，客户端会假设服务器知道它将新的 Bid 添加到了 Item#bids 集合并且该 Bid 必须被存储。你的服务器可以实现此功能，可能要使用该集合的 @OneToMany 映射上启用的合

并级联。然后，该服务的 `storeItem()` 方法会像上一节中那样工作，使用该 `Item` 并且调用 `ItemDAO` 让它(及其传递性依赖项)持久化。

这个应用程序中并非如此：服务器提供了一个独立的 `placeBid()` 方法。必须在出价被存储到数据库之前执行额外的验证，比如检查它是否比上一次出价高。还希望强制该 `Item` 版本的递增，以避免并发的出价。因此，要记录域模型实体关联的级联行为：`Item#bids` 并非传递性的，并且必须专门通过该服务的 `placeBid()` 方法来存储新的 `Bid` 实例。

服务器上 `placeBid()` 方法的实现会负责进行验证以及版本检查：

路径：/apps/app-stateless-server/src/main/java/org/jpwh/stateless/
AuctionServiceImpl.java

```
public class AuctionServiceImpl implements AuctionService {

    // ...

    @Override // 检查是否满足业务规则
    public Item placeBid(Bid bid) throws InvalidBidException {
        bid = bidDAO.makePersistent(bid);

        if (!bid.getItem().isValidBid(bid))
            throw new InvalidBidException("Bid amount too low!");

        itemDAO.checkVersion(bid.getItem(), true);

        return bid.getItem();
    }
}
```

这里会发生两件有趣的事情。首先，一个事务会被开启并且贯穿 `placeBid()` 方法的调用。对 `ItemDAO` 和 `BidDAO` 的嵌套 EJB 方法调用位于该相同的事务上下文中并且会继承该事务。对于持久化上下文来说也是如此：它具有与该事务相同的作用域③。这两个 DAO 类都会声明它们需要注入当前的 `@PersistenceContext`；运行时容器会提供绑定到当前事务的持久化上下文。使用无状态 EJB 进行事务和持久化上下文的创建以及传递非常简单，总是“伴随调用发生。”

其次，新 `Bid` 的验证是在域模型类中封装的业务逻辑。该服务会调用 `Item#isValid(Bid)` 并且将验证的职责委托给 `Item` 域模型类。这里是如何在 `Item` 类中实现这一点的代码：

路径：/apps/app-model/src/main/java/org/jpwh/model/Item.java

```
@Entity
public class Item implements Serializable {

    // ...

    public boolean isValidBid(Bid newBid) {
        Bid highestBid = getHighestBid();
        if (newBid == null)
            return false;
        if (newBid.getAmount().compareTo(new BigDecimal("0")) != 1)
            return false;
    }
}
```



```

    if (highestBid == null)
        return true;
    if (newBid.getAmount().compareTo(highestBid.getAmount()) == 1)
        return true;
    return false;
}
public Bid getHighestBid() {
    return getBids().size() > 0
        ? getBidsHighestFirst().get(0) : null;
}
public List<Bid> getBidsHighestFirst() {
    List<Bid> list = new ArrayList<>(getBids());
    Collections.sort(list);
    return list;
}
// ...
}

```

`isValid()`方法会执行几个检查来查明该 `Bid` 是否比上一次出价高。如果拍卖系统必须在某些时候支持“最低出价中标”的策略，那么你所必须做的就是修改 `Item` 域模型实现；使用该类的服务和 `DAO` 不会清楚其区别（显然，需要一条不同的消息用于 `InvalidBidException`）。

有争议的地方是，`getHighestBid()`方法的效率。它会将整个 `bids` 集合加载到内存中，在那里对其排序，然后只选用一个 `Bid`。一个优化过的变体看起来可能会像这样：

路径：/apps/app-model/src/main/java/org/jpwh/model/Item.java

```

@Entity
public class Item implements Serializable {

    // ...

    public boolean isValidBid(Bid newBid,
                             Bid currentHighestBid,
                             Bid currentLowestBid) {

        // ...

    }
}

```

该服务（或者你愿意的话，可以称之为控制器）仍旧完全不清楚任何业务逻辑——它无须知道一次新的出价是否必须比上一次出价高或低。该服务实现必须在调用 `Item#isValid()` 方法时提供 `currentHighestBid` 和 `currentLowestBid`。这是我们之前提示过的：你可能希望将操作添加到 `BidDAO`。可以用最有效的方式编写数据库查询来找到那些出价，而无须将所有的商品出价都加载到内存中并且在其中对它们进行排序。

该应用程序现在就完成了。它支持你打算实现的两个用例。让我们后退一步并且分析该结果。

18.2.3 分析无状态应用程序

已经实现了代码以便支持会话：从用户角度出发的工作单元。用户期望在一个工作流中执行一系列步骤，并且每个步骤都仅仅是临时的，直到他们用最后一步完成该会话。最后一步通常是从客户端到服务器的一个最后请求，它会结束会话。这听起来很像事务，但你可能必须在服务器执行几个系统事务来完成某个特定会话。问题是如何提供跨几个请求和系统事务的原子性。

用户的会话可能会具有任意程度的复杂性和持续时长。一个会话流程中的多个客户端请求都可以加载分离数据。因为你在控制客户端上的分离实例，所以如果在你的会话工作流中最后一步之前，你不对服务器上的任何实体实例进行合并、持久化或移除的话，你就可以轻易让一个会话变得原子化。在用户思考时间，在何处持有该组商品以便以某种方式对修改排队以及管理分离数据，这取决于你。只是在你确定希望“提交”该会话之前，不要从客户端调用任何对服务器进行永久变更的服务操作。

你必须关注的一个问题是分离引用的相等性：例如，如果加载几个 `Item` 实例并且将它们放在一个 `Set` 中或者将它们用作 `Map` 中的键时，就需要关注这个问题。因为之后你要在对象标识的受保障范围——持久化上下文——之外比较实例，你必须重写 `Item` 实体类上的 `equals()` 和 `hashCode()` 方法，就像 10.3.1 节中所阐释的一样。在只有一个分离的 `Item` 实例组的简单会话中，这不是必须的。你绝不会在一个 `Set` 中比较它们、将它们用作 `HashMap` 中的键，或者显式测试它们是否相等。

你应该为多用户应用程序启用 `Item` 实体的版本控制，就像第 11 章中“启用版本控制”一节所阐释的那样。在 `AuctionService#storeItem()` 中合并实体修改时，`Hibernate` 会递增 `Item` 的版本（不过，如果该 `Item` 没有被修改，则不会递增）。因此，如果另一个用户已经并发修改了一个 `Item` 的名称，那么 `Hibernate` 就会在提交该系统事务并且刷新该持久化上下文时抛出一个异常。使用这一乐观策略，第一个提交其会话的用户就总是会胜出。应该向第二个用户显示常见的错误消息：“对不起，有其他人修改了相同的数据；请重新开启你的会话。”

你已经创建的就是一个具有富客户端或胖客户端的系统；该客户端不是一个笨拙的输入/输出终端，而是一个具有独立于服务器的内部状态的应用程序（回想一下，服务器不会持有任何应用程序状态）。这样一个无状态服务器的其中一个优势在于，任何服务器都可以处理任意客户端请求。如果一个服务器出现故障，那么就可以将下一个请求路由到不同的服务器，并且会话处理将会继续。一个群集中的服务器不会共享任何东西；可以通过添加更多的服务器来轻易地横向扩展你的系统。显然，所有的应用程序服务器仍旧会共享数据库系统，但至少你仅要担心服务器这一层的扩展。

保持竞态条件之后的变更

在验收测试中，你很可能会发现用户不喜欢在检测到竞态条件时重启会话。他们可能需要悲观锁：当用户 A 编辑一个商品时，用户 B 甚至都不应该被允许将它加载到编辑器对话框中。根本性的问题并非会话结束时的乐观版本检查；而是你重新启动一个会话时会丢失你所有的工作。

不是呈现一条简单并发错误消息，而是可以提供一個对话框来允许用户保持他们现在

的无效变更，将其与其他用户做出的修改手动合并到一起，然后保存合并的结果。不过，要当心：实现这一功能可能会耗费时间，并且 Hibernate 不会提供太多帮助。

其缺点在于，需要编写富客户端应用程序，并且必须处理网络通信和数据序列化问题。复杂性从服务器端切换到了客户端，并且必须优化客户端和服务器之间的通信。

如果(JavaScript)客户端必须在几种 Web 浏览器上而不是在 EJB 客户端上运行，或在不同的(手机)操作系统上作为一个本地应用程序运行，那么这就肯定会是一个挑战。如果富客户端运行在流行的 Web 浏览器中，其中用户在每次访问你的网站时都会下载该客户端应用程序的最新版本，那么我们推荐这个架构。在几个平台上推出本地客户端，并且维护和升级这些安装程序，会是一个巨大的负担，即便是对于可以控制用户环境的中等大小的内网来说，也是如此。

不使用 EJB 环境，就必须自定义序列化以及客户端和服务器之间分离实体状态的传递。你能够序列化与反序列化一个 Item 实例吗？如果没有用 Java 编写你的客户端会发生什么？我们将在 19.4 节中探讨这个问题。

接下来，你要再次实现相同的用例，但使用一种非常不同的策略。服务器现在将持有应用程序的会话状态，而客户端将仅仅是一个笨拙的输入/输出设备。这就是一种具有一个瘦客户端和一个状态服务器的架构。

18.3 构建一个状态服务器

你要编写的应用程序仍然很简单。它像之前一样支持相同的两个用例：编辑一个拍卖商品以及为一个商品放置出价。对于该应用程序的用户来说没有可见的区别；EJB 控制台客户端看起来仍旧像图 18-2 和图 18-4 一样。

使用瘦客户端，将输出的数据转换成被瘦客户端理解的显示格式就是服务器的任务——例如，转换成由 Web 浏览器呈现的 HTML 页面。该客户端会将用户输入操作直接传递给服务器——例如，就像简单 HTML 表单提交一样。服务器要负责解码以及将输入转换成更高级别的域模型操作。不过，目前我们要简化这一部分，并且仅用 EJB 客户端进行远程方法调用。

然后服务器还必须持有会话数据，通常是存储在与某特定客户端关联的某种服务器端会话中。注意，一个客户端会话的作用域比单个会话的作用域要大；用户可以在一个会话期间执行多个会话。如果用户离开客户端并且未完成一次会话，就必须在某些时候在服务器上移除临时会话数据。服务器通常会用超时来处理这种情况；例如，服务器可能会在某个不活动周期之后丢弃一个客户端的会话以及它包含的所有数据。这听起来很像 EJB 状态性会话 beans 的一个任务，并且，如果正好需要一个标准解决方案的话，那么它们确实适用于此类架构。

要牢记那些根本性问题，下面让我们实现第一个用例：编辑一个拍卖商品。

18.3.1 编辑一个拍卖商品

客户端同样为用户呈现一组拍卖商品，并且用户要选取一个。应用程序的这个部分很简单，并且不必在服务器上持有任何会话状态。看看图 18-6 中的调用序列和上下文。

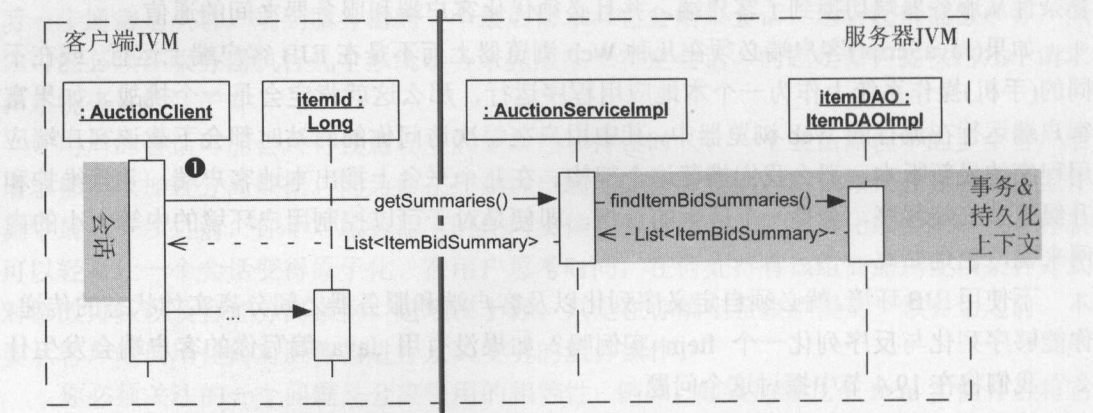


图 18-6 客户端检索用于即时显示的已经被转换过的数据

由于该客户端不是非常智能，因此它不会也不应该理解一个 Item 实体类是什么。它会加载 ❶ ItemBidSummary 数据传输对象的一个 List：

```
路径: /apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/
AuctionServiceTest.java
```

```
List<ItemBidSummary> itemBidSummaries = auctionService.getSummaries();
```

服务器会用一个无状态组件实现这个任务，因为它此时无须持有任何会话状态：

```
路径: /apps/app-stateful-server/src/main/java/org/jpwh/stateful/
AuctionServiceImpl.java
```

```
@javax.ejb.Stateless
@javax.ejb.Local(AuctionService.class)
@javax.ejb.Remote(RemoteAuctionService.class)
public class AuctionServiceImpl implements AuctionService {

    @Inject
    protected ItemDAO itemDAO;

    @Override
    public List<ItemBidSummary> getSummaries() {
        return itemDAO.findItemBidSummaries();
    }
}
```

即使你有一个状态性服务器架构，你的应用程序中也会有许多无须在服务器上持有任何状态的简短会话。这很正常，也很重要：在服务器上持有状态会消耗资源。如果使用一个状态性 bean 实现 getSummaries()操作，那么就是在浪费资源。你仅会将状态性 bean 用于

单个操作，然后它会消耗内存，直到容器让其过期。状态性服务器架构并不意味着你只能使用状态性服务器端组件。

接下来，客户端会呈现 ItemBidSummary 列表，它只包含每个拍卖商品的标识符、其描述以及当前最高出价。这正是用户在屏幕上所看到的内容，如图 18-2 所示。然后用户要输入一个商品标识符并且开启一个会话来处理这个商品。可以在图 18-7 中看到此会话的路线图。

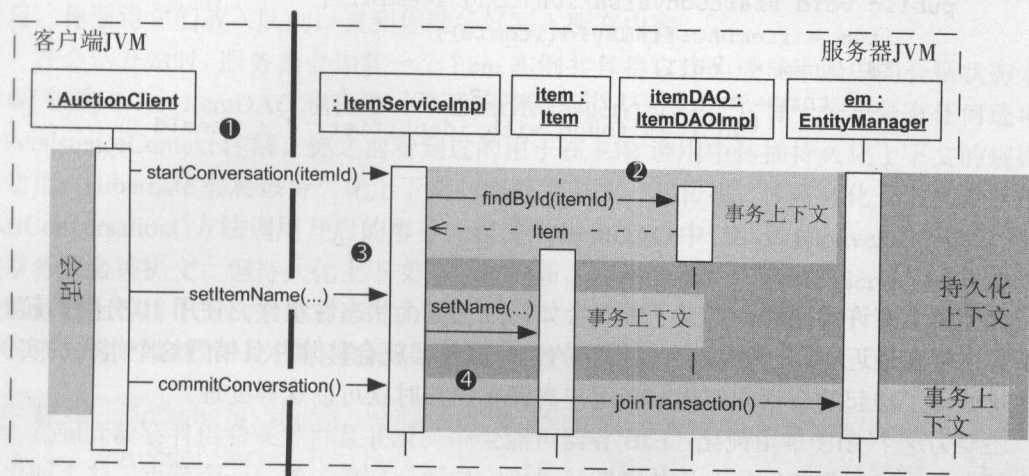


图 18-7 客户端控制服务器上的会话边界

客户端会通过传递一个商品标识符值❶来通知服务器现在应该开启一个会话：

```
路径: /apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/
AuctionServiceTest.java
```

```
itemService.startConversation(itemId);
```

此处调用的服务不是上一节中的共享无状态 AuctionService。这个新的 ItemService 是一个状态性组件；服务器将创建一个实例并且将其独占式指定给这个客户端。要使用一个状态性会话 bean 来实现这个服务：

```
路径: /apps/app-stateful-server/src/main/java/org/jpwh/stateful/
ItemServiceImpl.java
```

```
@javax.ejb.Stateful(passivationCapable = false)
@javax.ejb.StatefulTimeout(10)           ← 分钟
@javax.ejb.Local(ItemService.class)
@javax.ejb.Remote(RemoteItemService.class)
public class ItemServiceImpl implements ItemService {

    @PersistenceContext(type = EXTENDED, synchronization = UNSYNCHRONIZED)
    protected EntityManager em;

    @Inject
    protected ItemDAO itemDAO;
```

```

@Inject
protected BidDAO bidDAO;

// Server-side conversational state
protected Item item;
// ...

@Override
public void startConversation(Long itemId) {
    item = itemDAO.findById(itemId);
    if (item == null)
        throw new EntityNotFoundException(
            "No Item found with identifier: " + itemId
        );
}
// ...
}

```

这个类上有许多注解，定义了容器会如何处理这个状态性组件。使用 10 分钟的超时，如果客户端在最近 10 分钟内没有调用它的话，服务器就会移除并且销毁这个组件的实例。这样就处理了挂起的会话：例如，当用户离开客户端时就可以这样处理。

还要为这个 EJB 禁用钝化：EJB 容器可能会序列化并且将状态性组件存储到硬盘，以便保留内存或者在需要进行会话故障转移时将它传递到群集中的另一个节点。这个钝化不会生效，因为有一个成员字段：EntityManager。你在使用 EXTENDED 切换将持久化上下文附加到这个状态性组件，并且 EntityManager 不是 java.io.Serializable 的。

常见问题解答：为何不能序列化一个 EntityManager

持久化上下文和 EntityManager 不能被序列化没什么技术性的原因可言。当然，EntityManager 必须在反序列化时将其自身重新附着到目标机器上的正确 EntityManagerFactory，但那只是一个实现细节。目前持久化上下文的钝化已经超出了 JPA 和 Java EE 规范的范畴。

不过，大多数供应商都提供了一个可以序列化并且知道如何正确反序列化自身的实现。Hibernate 的 EntityManager 可以被序列化和反序列化，并且当它必须在反序列化时找到正确持久化单元时，它会尝试变得足够智能。

有了 Hibernate 和一台 Wildfly 服务器，你就可以在上一个示例中启用钝化，并且你会得到会话故障转移以及使用状态性服务器的高可用性，还有扩展的持久化上下文。不过，这并非标准化的；并且我们稍后将谈到，这个策略很难扩展。

也会出现甚至 Hibernate 的 EntityManager 都不能被序列化的情况。你可能会遇到尝试绕过这一问题的遗留框架代码，比如 Seam 框架的 ManagedEntityInterceptor。你应该避免这种情况并且寻求一个更为简单的解决方案，这通常意味着在群集、无状态服务器设计，或者我们将在下一章中与请求作用域持久化上下文一起探讨的 CDI 服务器端会话管理解决方案中使用粘滞会话。

你要使用 @PersistenceContext 来声明此状态性 bean 需要一个 EntityManager，并且容器应该扩展该持久化上下文来跨越像状态性会话 bean 的生命周期一样的相同周期。这个扩展

的模式是状态性 EJB 的一个专有选项。如果不使用它的话，容器将会在事务提交的任何时候创建和关闭一个持久化上下文。这里，你希望持久化上下文在所有事务边界之外保持开放并且仍然附加到状态性会话 bean 实例。

此外，你也不希望持久化上下文在事务提交时被自动刷新，因此要将它配置为 UNSYNCHRONIZED。Hibernate 将只会在你手动将持久化上下文联结到一个事务之后才刷新它。现在 Hibernate 不会自动将你对已加载的持久化实体实例所做的变更写入到数据库；相反，你要将它们放入队列，直到你准备好写入所有内容。

在会话开始时，服务器会加载一个 Item 实例并且将它作为成员字段中的会话状态来保持❷(见图 18-7)。ItemDAO 也需要一个 EntityManager；记住，它有一个不带有任何选项的 @PersistenceContext 注解。你之前看到过的用于在 EJB 调用中传播持久化上下文的规则仍然适用：Hibernate 会将该持久化上下文与事务上下文一起传播。该持久化上下文会伴随为 startConversation()方法调用开启的事务一起进入 ItemDAO 中。当 startConversation()返回时，该事务就会被提交，但持久化上下文不会被刷新，也不会被关闭。ItemServiceImpl 实例会在服务器上等待下一次来自客户端的调用。

来自客户端的下一调用会指示服务器修改该商品的名称❸：

路径：/apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/
AuctionServiceTest.java

```
itemService.setItemName("Pretty Baseball Glove");
```

在服务器上，会为 setItemName()方法开启一个事务。但由于其中没有涉及任何事务资源(没有 DAO 调用，没有 EntityManager 调用)，所以什么都不会发生，只会对你在会话状态中持有的 Item 进行一次修改：

路径：/apps/app-stateful-server/src/main/java/org/jpwh/stateful/
ItemServiceImpl.java

```
public class ItemServiceImpl implements ItemService {  
    // ...  
    @Override  
    public void setItemName(String newName) {  
        item.setName(newName);  
    }  
    // ...  
}
```

在这个方法的事务提交时，不会刷新持久化上下文；它是非同步的。

注意，该 Item 仍然处于持久化状态——持久化上下文仍旧是开启的！不过由于它并非同步的，因此它不会检测你对 Item 做出的修改，因为在事务提交时它不会被刷新。

最后，客户端会提供 OK 来存储服务器上的所有变更以结束该会话(见图 18-7)❹：

路径: /apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/
AuctionServiceTest.java

```
itemService.commitConversation();
```

在服务器上, 可以将变更刷新到数据库并且丢弃该会话状态:

路径: /apps/app-stateful-server/src/main/java/org/jpwh/stateful/
ItemServiceImpl.java

```
public class ItemServiceImpl implements ItemService {
```

```
    @Override
```

```
    @javax.ejb.Remove
```

在方法完成之后移除 bean。

```
    public void commitConversation() {
```

```
        em.joinTransaction();
```

```
    }
```

```
}
```

持久化上下文被联结到当前事务, 并且在这个方法返回时被刷新, 以保存变更。

你现在已经完成了第一个用例的实现。我们将跳过第二个用例(放置一个出价)的实现并且请你参考示例代码以了解详细细节。第二个用例的代码几乎与第一个相同, 并且要理解它, 你应该没什么问题。你必须理解的一个重要方面是, 持久化上下文和事务处理在 EJB 中如何运行。

提示: 对于在不同类型的组件之间传播持久化上下文, 还有额外的一些规则适用于 EJB。它们非常复杂, 并且在实践中我们还从来没有看到过任何合适的使用它们的用例。例如, 你可能不会从一个无状态 EJB 调用一个状态性 EJB。另一个复杂之处在于, 已禁用的或可选的 EJB 方法事务, 它们也会影响持久化上下文与组件调用一起传播的方式。我们在本书的上一版中阐释过这些传播规则。我们建议你尝试仅适用本章中介绍的策略, 并且让事情保持简单。

我们来探讨状态性和无状态服务器架构之间的一些区别。

18.3.2 分析状态性应用程序

就像我们对于无状态应用程序的分析一样, 第一个问题是从用户角度看, 工作单元是如何实现的。实际上, 需要弄明白会话的原子性是如何发挥作用的, 以及你如何才能让工作流中的所有步骤看起来像单个单元。

有些时候, 通常是会话中的最后一个请求发生时, 你会提交会话并且将变更写入到数据库。如果在会话的最后一个事件之前没有用事务将扩展的 EntityManager 联结起来, 则该会话就是原子的。如果仅仅在非同步模式中读取数据, 就不会发生脏检查和刷新。

当保持持久化上下文开启时, 可以通过访问代理和未加载的集合来保持延迟加载数据; 这显然是很方便的。不过, 如果用户需要较长时间来触发下一个请求, 则已加载的 Item 和其他数据将会过期。如果需要来自数据库的一些更新, 那么可能会希望在会话期间 refresh() 一些托管实体实例, 就像 10.2.6 节中所阐释的那样。或者, 可以刷新以便撤销会话

期间的操作。例如,如果用户在对话框中修改了 `Item#name`,但之后决定撤销这次修改,就可以 `refresh()` 该持久化 `Item` 实例以便从数据库中检索“旧”名称。这是已扩展持久化上下文的一个非常棒的功能,它允许 `Item` 在托管持久化状态中总是可用。

会话中的存储点

你可能很熟悉 JDBC 事务中的存储点:在一个事务中修改了一些数据之后,你要设置一个存储点;稍后将事务回滚到该存储点,以撤销一些工作但保持设置该存储点之前做出的变更。遗憾的是, Hibernate 没有为持久化上下文提供类似于存储点的概念。你只能使用 `refresh()` 操作将实体实例回滚到其数据库状态。一个事务中的常规 JDBC 存储点可以与 Hibernate 一起使用(需要一个 `Connection`; 参阅 17.1 节),但它们不会帮助你实现会话中的撤销。

状态性服务器架构可能更难水平扩展。如果一台服务器出现故障,那么当前会话的状态并且实际上整个会话都会丢失。在几台服务器上复制会话是开销很大的操作,因为一台服务器上会话数据的任何修改都涉及与(可能是所有)其他服务器的网络通信。

使用状态性 EJB 和成员扩展的 `EntityManager`,这个扩展的持久化上下文的序列化就是不可行的。如果在一个群集中使用状态性 EJB 和一个扩展的持久化上下文,则要考虑启用粘滞会话时,会造成特定客户端请求总是路由到相同服务器的问题。这可以使用额外的服务器轻易处理更多的负载,但你的用户必须接受一台服务器出现故障时会丢失会话状态这一结果。

另一方面,状态性服务器可以使用其扩展的持久化上下文在用户会话中充当第一道缓存线。一旦为某个用户会话加载了一个 `Item`,就不会在相同会话中从数据库再次加载这个 `Item`。这会是降低数据库服务器(扩展时成本最高的层)负载的一个伟大工具。

相较于仅持有分离实例,扩展的持久化上下文策略对于服务器的内存需求更大: Hibernate 中的持久化上下文包含所有托管实例的快照副本。你可能希望手动 `detach()` 托管实例,以便控制在持久化上下文中保持什么内容,或者禁用脏检查和快照(同时仍旧可以延迟加载),就像 10.2.8 节中所阐释的那样。

当然,也有瘦客户端和状态性服务器的可替代实现。可以在服务器上手动使用常规的请求作用域持久化上下文并且管理分离的(非持久化的)实体实例。这对于分离与合并肯定可行,但是会需要大量的工作量。扩展的持久化上下文的一个主要优势,即便是跨请求的透明延迟加载,也都将不再可用。在下一章中,我们将介绍在 CDI 和 JSF 中使用请求作用域持久化上下文的这样一种状态性服务的实现,可以将其与本章介绍的 EJB 的扩展持久化上下文功能做比较。

瘦客户端系统通常会比富客户端在服务器上产生的负载更大。用户每次与应用程序交互时,一个客户端事件就会产生一次网络请求。一个 Web 应用程序中的每次鼠标单击甚至都可能发生这种情况。只有服务器知道当前会话的状态,并且必须准备以及呈现用户浏览的所有信息。另一方面,富客户端可以在一次请求中加载一个会话所需的原始数据,对数据进行转换,并且按需将数据本地绑定到用户界面。富客户端中的对话框可以在客户端对修改进行排队并且在会话结束必须让变更持久化时才触发一次网络请求。

使用瘦客户端的另外一个挑战是,一个用户的平行会话:如果一个用户同时编辑两个

商品会发生什么——例如，在两个 Web 浏览器选项卡中？这意味着该用户与服务器有两个平行会话。服务器必须根据会话分离用户会话中的数据。因此，一个会话期间的客户端请求必须包含某种会话标识符，以便可以为每个请求从用户的会话中选择正确的会话状态。这在 EJB 客户端和服务端中会自动进行，但可能没有内置到你喜欢的 Web 应用程序框架中（除非你喜欢使用 JSF 和 CDI，下一章将会对它们进行介绍）。

状态性服务器的一个明显好处是，较少依赖客户端平台；如果客户端是一个具有很少活动部件的简单输入/输出终端，那么事情将很少会出错。必须实现数据验证和安全性检查的唯一位置就是服务器。没有要处理的部署问题；可以在服务器上进行应用程序升级而无须触及客户端。

如今，瘦客户端系统的优势很少，并且状态性服务器的安装使用都下降了。在 Web 应用程序领域尤为如此，其中方便的扩展性常常是一个主要的关注点。

18.4 本章小结

- 实现了简单会话——从应用程序用户角度来看的工作单元。
- 你看到了两种服务器和客户端设计，分别使用了无状态和状态性服务器，并且学习了 Hibernate 如何适用于这两种架构。
- 可以使用分离的实体状态或者扩展的会话作用域持久化上下文。

构建 Web 应用程序

第 19 章

19

本章内容简介：

- 集成 JPA 与 CDI 和 JSF
- 在表格中浏览数据
- 实现长期运行的会话
- 自定义实体序列化

在本章中，你将看到 Hibernate 如何在一个典型的 Web 应用程序环境中运行。用于 Java 的 Web 应用程序框架有数十种，因此，如果在此我们没有对于你喜欢的组合进行介绍，那么我们表示歉意。我们将探讨标准 Java 企业版环境中的 JPA，尤其是会结合标准上下文和依赖注入(CDI)、JavaServer Faces(JSF)以及用于 RESTFUL Web 服务(JAX-RS)的 Java API 进行介绍。一如既往，我们将向你介绍可以在其他专有环境中应用的模式。

首先我们回到持久化层并且介绍用于 DAO 类的 CDI 管理。然后我们使用一个通用的解决方案扩展这些类以便对数据进行排序和分页。这个解决方案必须在表格中显示数据的任何时候都很有用，无论你选择何种框架。

接下来，要在持久化层之上编写一个完全的功能性 JSF 应用程序，并且查看 Java EE 会话作用域，其中 CDI 和 JSF 会协同工作以便为服务器端组件提供一个简单的状态模型。如果不喜欢上一章中介绍的具有扩展持久化上下文的状态性 EJB，那么可能服务器上这些具有分离实体状态的会话示例正是你所寻求的。

最后，如果更愿意编写具有富客户端、无状态服务器以及像 JAX-RS、GWT 或 AngularJS 这样的框架的 Web 应用程序，那么我们将介绍如何用 XML 和 JSON 格式自定义 JPA 实体实例的序列化。我们首先介绍将持久化层从 EJB 组件移植到 CDI。

19.1 集成 JPA 与 CDI

CDI 标准在 Java EE 运行时环境中提供了类型安全的依赖注入以及组件生命周期管理系统。你在上一章中看到过 `@Inject` 注解，并且使用它来将 `ItemDAO` 和 `BidDAO` 组件与服务 EJB 类连接到一起。

在 DAO 类内部使用的 JPA `@PersistenceContext` 注解只是另一个特殊的注入用例：告

知运行时容器提供以及自动处理一个 `EntityManager` 实例。这是一个容器托管的 `EntityManager`。不过，其中也附加了一些字符串，比如我们在上一章中探讨过的持久化上下文传播以及事务规则。当所有的服务和 DAO 类都是 EJB 的时候，这样的规则就很方便，但如果不使用 EJB，则可能不希望遵循这些规则。使用一个应用程序托管的 `EntityManager`，就能创建你自己的持久化上下文管理、传播以及注入规则。

现在要将这些 DAO 类重写为简单的 CDI 托管 beans，它们就像 EJB 一样：具有额外注解的普通 Java 类。希望对一个 `EntityManager` 执行 `@Inject` 操作并且删除 `@PersistenceContext` 注解，并因而得到对持久化上下文的全面控制。在可以注入你自己的 `EntityManager` 之前，必须生成它。

19.1.1 生成一个 `EntityManager`

CDI 术语中的生产者是一个工厂，它被用于实例的自定义创建并且告知运行时容器在应用程序需要基于所声明作用域的实例时调用一个自定义例程。例如，容器仅会在应用程序的生命周期期间创建一次该应用程序作用域的实例。该容器会为每个由一台服务器处理的请求创建一个请求作用域实例，并且为用户与服务器的每个会话创建一个会话作用域实例。

CDI 规范将请求和会话的抽象概念映射到了 `Servlet` 的请求和会话。记住，JSF 与 JAX-RS 都构建在 `Servlet` 上，因此 CDI 能很好地适用于那些框架。换句话说，不要过于担心这一点：在 Java EE 环境中，所有的集成工作都已经为你完成了。

下面创建一个请求作用域的 `EntityManager` 实例的生产者：

路径：/apps/app-web/src/main/java/org/jpwh/web/dao/EntityManagerProducer.
java

```
@javax.enterprise.context.ApplicationScoped  ← ① 仅需要 1 个生产者
public class EntityManagerProducer {

    @PersistenceUnit  ← ② 得到持久化单元
    private EntityManagerFactory entityManagerFactory;

    @javax.enterprise.inject.Produces  ← ③ 得到 EntityManager
    @javax.enterprise.context.RequestScoped
    public EntityManager create() {
        return entityManagerFactory.createEntityManager();
    }

    public void dispose(
        @javax.enterprise.inject.Disposes  ← ④ 关闭持久化上下文
        EntityManager entityManager) {
        if (entityManager.isOpen())
            entityManager.close();
    }
}
```


- ❶ 此 CDI 注解声明，整个应用程序中只需要一个生产者：永远只有一个 `EntityManagerProducer` 实例。
- ❷ Java EE 运行时会为你提供在 `persistence.xml` 中配置的持久化单元，它也是一个应用程序作用域的组件(如果单独并且在 Java EE 环境之外使用 CDI，则可以转而使用静态的 `Persistence.createEntityManagerFactory()` 引导程序)。
- ❸ 无论何时需要一个 `EntityManager`，都会调用 `create()`。容器会在由服务器处理的一个请求期间重用相同的 `EntityManager`(如果忘记在该方法上使用 `@RequestScoped`，`EntityManager` 就会被限制于应用程序作用域，就像生产者类一样！)
- ❹ 在一个请求结束并且该请求上下文被销毁时，CDI 容器会调用这个方法去去除 `EntityManager` 实例。你创建了这个应用程序托管的持久化上下文(参阅 10.1.2 节)，因此关闭它是你的责任。

使用 CDI 注解的类的一个常见问题是，注解的错误导入。在 Java EE 7 中，有两个注解都被称为 `@Produces`；另一个是 `javax.ws.rs (JAX-RS)`。这与 CDI 生产者注解不同，并且如果选取了错误的一个，就可能会花费数小时在代码中排查这个错误。另一个重复项是 `@RequestScoped`，也位于 `javax.faces.bean(JSF)` 中。就像 `javax.faces.bean` 包中所有其他过时的 JSF bean 管理注解一样，在有更现代的 CDI 可用时不要使用它们。我们希望未来的 Java EE 规范版本会解决这些歧义。

现在有了一个应用程序托管的 `EntityManager` 的生产者以及请求作用域的持久化上下文。接下来，必须找到一个办法让 `EntityManager` 获悉与你的系统事务有关的信息。

19.1.2 将 EntityManager 与事务联结起来

当服务器必须处理一个 `servlet` 请求时，容器会在首次需要 `EntityManager` 以用于注入时自动创建它。记住，如果一个事务已经在进程中，那么手动创建的 `EntityManager` 将仅自动联结该系统事务。否则，它将是非同步的：你要在自动提交模式中读取数据，且 `Hiernate` 不会刷新该持久化上下文。

容器调用 `EntityManager` 生产者的情况或者在一次请求期间正好发生首个 `EntityManager` 注入的情况并不总是显而易见的。当然，如果不使用一个系统事务来处理请求，那么得到的 `EntityManager` 就总是非同步的。因此，必须确保 `EntityManager` 知道你有一个系统事务。

要在超接口中将用于此目的的一个方法添加到持久化层：

路径：/apps/app-web/src/main/java/org/jpwh/web/dao/GenericDAO.java

```
public interface GenericDAO<T, ID extends Serializable>
    extends Serializable {

    void joinTransaction();

    // ...

}
```

当确定你正处于一个事务中时，应该在存储数据之前在任何 DAO 上调用这个方法。如果

忘记了, Hibernate 就会在你尝试写入数据时抛出一个 `TransactionRequiredException` 异常, 表明 `EntityManager` 在事务开启之前已经被创建了并且它未收到与该事务有关的信息。如果想要练习你的 CDI 技能, 则可以尝试使用一个 CDI 修饰器或拦截器来实现这方面的练习。

我们来实现这个 `GenericDAO` 接口方法并且将新的 `EntityManager` 连接到 DAO 类。

19.1.3 注入一个 `EntityManager`

老的 `GenericDAOImpl` 依赖 `@PersistenceContext` 注解以在一个字段中注入 `EntityManager`, 或者在使用 DAO 之前需要有人调用 `setEntityManager()`。有了 CDI, 就可以使用较为安全的构造函数注入技术:

路径: `/apps/app-web/src/main/java/org/jpwh/web/dao/GenericDAOImpl.java`

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    protected final EntityManager em;
    protected final Class<T> entityClass;

    protected GenericDAOImpl(EntityManager em, Class<T> entityClass) {
        this.em = em;
        this.entityClass = entityClass;
    }

    public EntityManager getEntityManager() {
        return em;
    }

    @Override
    public void joinTransaction() {
        if (!em.isJoinedToTransaction())
            em.joinTransaction();
    }

    // ...
}
```

无论是谁想要实例化一个 DAO 都必须提供一个 `EntityManager`。类的不变量的这一声明是一个更为健壮的保障; 因此, 尽管我们常常在示例中使用字段注入, 但应该总是首先考虑使用构造函数注入(我们在有些示例中没有这样做, 是因为它们可能会很长, 而这本书已经很厚重了)。

在具体的(实体 DAO)子类中, 要在构造函数上声明必要的注入:

路径: `/apps/app-web/src/main/java/org/jpwh/web/dao/ItemDAOImpl.java`

```
public class ItemDAOImpl
    extends GenericDAOImpl<Item, Long>
    implements ItemDAO {

    @Inject
    public ItemDAOImpl(EntityManager em) {
```

```
super(em, Item.class);
```

```
// ...
```

当应用程序需要一个 ItemDAO 时，CDI 运行时会调用 EntityManagerProducer，然后调用 ItemDAOImpl 构造函数。在某个特定请求期间，容器将为所有 DAO 中的任何注入重用相同的 EntityManager。

那么 ItemDAO 的作用域是什么？由于没有为实现类声明一个作用域，因此它是从属的。无论何时谁需要 ItemDAO，它会被创建，之后该 ItemDAO 实例会位于相同上下文中，并且作用域与其调用方相同，从属于该调用对象。这对于持久化层 API 来说是一个好的选择，因为你将作用域决策委托给了使用服务调用该持久化层的更上层。

常见问题解答：我要如何使用 CDI 处理多个持久化单元？

如果有多个数据库——多个持久化单元——就可以使用 CDI 限定符来区分它们。限定符是一个自定义注解。要编写该注解，比如 @BillingDatabase，并且将它标记为一个限定符。然后要将它放置到从特定持久化单元创建 EntityManager 的方法上的 @Produces 旁边。现在任何想要这个 EntityManager 的人也都要将 @BillingDatabase 添加到 @Inject 旁边。

现在就准备好在服务类中对一个 ItemDAO 字段执行 @Inject 操作。在使用启用 CDI 的持久化层之前，我们来添加一些用于数据分页和排序的功能。

19.2 数据的分页和排序

一个非常常见的任务是使用一个查询从数据库加载数据，然后在网页的表格中显示这些数据。还必须经常实现数据的动态分页和排序：

- 由于查询返回的信息要比可以在单个页面上显示的信息多得多，因此只能显示数据的一个子集。只能在数据表格中呈现特定数量的行，并且让用户可以选择来到这些行的下一页、上一页、第一页或最后一页。用户还会期望在他们切换页面时应用程序保留排序。
- 用户希望能够单击表格中的一列标题并且根据这个列的值对表格的行进行排序。

通常，可以按照升序或者降序进行排列；这可以由后续的列标题单击来切换。

现在你实现了一个用于通过数据页面进行浏览的通用解决方案，这是基于 JPA 提供的持久化类的元模型来实现的。

页面浏览的实现可以有两种变化：使用偏移量或者搜寻技术。我们先来看看两者的区别以及你希望实现什么。

19.2.1 偏移量分页与搜寻分页对比

图 19-1 显示了使用基于偏移量分页机制的浏览 UI 的一个示例。你看到这里少数拍

卖商品并且使用了三条记录的这一较小的页面大小。此处，你正在第一页；应用程序会动态呈现到其他页面的链接。当前排列顺序是按照商品名称升序排列。可以单击列标题并且将排序修改为按名称、拍卖结束日期或最高出价金额降序(或升序)排列。单击每个表格行中的商品名称会打开该商品的详细信息视图，其中可以对一个商品进行出价。我们在本章稍后将使用这个用例。

Catalog		
< < Items 1 to 3 of 7 > >		
Item ↑	Auction End	Highest Bid
Aquarium	07. Mar 2018 15:00	-
Baseball Glove	06. Mar 2018 14:00	13.00
Coffee Machine	10. Mar 2018 10:11	6.00

图 19-1 通过偏移量浏览分类页面

此页面背后是使用基于行数的偏移量和限制条件的数据库查询。用于此页面的 Java 持久化 API 是 `Query#setFirstResult()` 和 `Query#setMaxResults()`，我们在 14.2.4 节中探讨过它们。你要编写一个查询，然后让 `Hibernate` 围绕它包装偏移量和限制子句，这取决于数据库 SQL 方言。

现在思考一下替代项：使用搜寻方法的分页，如图 19-2 所示。这里没有给用户提供选项来根据偏移量跳转到任意页面；只允许他们向前搜寻到下一个页面。这可能看起来有些受限，但在需要循环滚动时，可能已经看到甚至实现了这样一个分页例程。例如，可以在用户到达表格/屏幕底部时自动加载并且显示数据的下一页。

Catalog		
< < Items 1 to 3 of 7 > >		
Item ↑	Auction End	Highest Bid
Aquarium	07. Mar 2018 15:00	-
Baseball Glove	06. Mar 2018 14:00	13.00
Coffee Machine	10. Mar 2018 10:11	6.00
Total: 7		Next page...

图 19-2 通过搜寻下一页浏览分类

搜寻方法依赖查询中的特殊额外限制来检索数据。当必须加载下一个页面时，就是在查询名称“大于[Coffee Machine]”的所有商品。你不是通过使用 `setFirstResult()` 的结果行偏移量来向前搜寻，而是通过基于一些键的排序值限制结果来搜寻的。就算你不熟悉搜寻分页(有时也称为键集分页)，我们也确信当看到本节稍后内容中的查询时并不会觉得难以理解。

我们来比较这两种技术的优势和劣势。当然可以使用偏移量分页或使用搜寻技术的直接页面导航来实现一个循环滚动功能；但它们各自有其自身的长处和短处：

- 当用户希望直接跳转到某些页面时，偏移量方法就很合适。例如，许多搜索引擎都提供了直接跳转到查询结果第 42 页或直接到最后一页的选项。由于可以根据期望的页码轻松计算偏移量以及行范围的限制，因此可以毫不费力地实现它。使用搜寻方法，要提供这样的页面跳转 UI 就更为困难了；必须知道要搜寻的值。你并不清楚正好在第 42 页之前客户端所显示的商品名称是什么，因此你无法向前搜寻名称为“大于 X”的商品。搜寻最适合于用户仅通过数据列表或表格逐页向前或向后浏览的 UI，并且你要能够轻易记住显示给用户的最后一个值。
- 搜寻的一个很好的用例就是基于你无须记住的锚点值进行分页。例如，所有姓名以 C 开头的顾客可以在一个页面上，而那些以 D 开头的在下一个页面上。或者，每个页面都显示已经到达其最高出价金额的特定阈值的拍卖商品。
- 如果抓取较大的页面数量，那么偏移量方法的执行将会非常糟糕。如果跳转到第 5000 页，则数据库必须计算所有的行并且在它可以跳过数据的 4999 页之前就准备好第 5000 页数据，以便向你显示结果。一个常用的变通方法是限制用户可以跳转的页数：例如，仅允许直接跳转到前 100 页并且强制用户调整查询限制以便得到一个较小的结果。搜寻方法通常比偏移量方法要快，即便是在页面数量较少的情况下也是如此。数据库查询优化器可以直接跳转到期望页面的起始位置并且有效限制要扫描的索引范围。在之前页面上显示的记录无须被考虑或计数。
- 偏移量方法有时可能会显示不正确的结果。尽管该结果可能与数据库中的数据是一致的，但你的用户可能还是会认为它不正确。当应用程序在用户浏览时候插入新的记录或者删除已有记录时，就可能出现异常情况。想象一下，用户正在查看第 1 页，而有人添加了一行会出现在第 1 页上的新记录。如果该用户现在检索第 2 页，那么他们在第 1 页上可能已经看过的一些记录就会向前推送到第 2 页上。如果第 1 页上的一条记录被删除了，那么用户可能就会在第 2 页上错过一条数据，因为有一条记录已经被推送回第 1 页了。搜寻方法可以避免这些异常情况；记录不会神秘地再次出现或消失。

下面我们要介绍如何通过扩展持久化层来实现这两种分页技术。首先你要处理持有当前分页以及数据表格视图排序设置的一个简单模型。

19.2.2 在持久化层中分页

当希望调整查询和呈现数据页面时，需要保留一些与页面大小以及当前正在浏览的页面有关的详情。下面是封装了此信息的一个简单模型类；它是一个抽象超类，可同时用于偏移量和搜寻技术：

路径：/apps/app-web/src/main/java/org/jpwh/web/dao/Page.java

```
public abstract class Page {  
    public static enum SortDirection {  
        ASC,  
        DESC  
    }  
}
```

```

                                ❶显示所有记录
protected int size = -1;
                                ❷保留记录计数
protected long totalRecords;
                                ❸对记录排序
protected SingularAttribute sortAttribute;
protected SortDirection sortDirection;
                                ❹限制可排序属性
protected SingularAttribute[] allowedAttributes;

// ...

abstract public <T> TypedQuery<T> createQuery(
    EntityManager em,
    CriteriaQuery<T> criteriaQuery,
    Path attributePath
);

```

- ❶ 该模型会保留每个页面的大小以及每个页面显示的记录数量。-1 这个值是特殊的，它意味着“没有限制；显示所有记录。”
- ❷ 保留总记录的数量对于某些计算来说是必要的：例如，在要确定是否有“下一页”的时候。
- ❸ 分页总是需要一个确定的记录顺序。通常，要根据实体类的某个属性按照升序或降序来排序。javax.persistence.metamodel.SingularAttribute 是 JPA 中一个实体或者一个可嵌入类的属性；它不是一个集合(不能在查询中“根据集合来排序”)。
- ❹ 在创建页面模型时会设置 allowedAttributes 列表。它会将可能的可排序属性限制为可以在查询中处理的那些属性。

我们没有显示出来的 Page 类的一些方法很简单——主要是获取方法和设置方法。不过，createQuery() 这个抽象方法是子类必须实现的：它与在执行查询前分页设置如何被应用到 CriteriaQuery 有关。

首先要在持久化层中引入这个 Page 接口。在 DAO 接口中，API 会接受一个 Page 实例，在其中希望支持逐页检索数据：

路径：/apps/app-web/src/main/java/org/jpwh/web/dao/ItemDAO.java

```

public interface ItemDAO extends GenericDAO<Item, Long> {

    List<ItemBidSummary> getItemBidSummaries(Page page);

    // ...
}

```

你希望呈现的数据表格显示了 ItemBidSummary 数据传输对象的一个 List。这个示例中查询的结果并不重要；可以像检索一系列 Item 实例那样检索它。这是 DAO 实现的一个简短摘要：

路径: /apps/app-web/src/main/java/org/jpwh/web/dao/ItemDAOImpl.java

```
public class ItemDAOImpl
    extends GenericDAOImpl<Item, Long>
    implements ItemDAO {

    // ...

    @Override
    public List<ItemBidSummary> getItemBidSummaries(Page page) {
        CriteriaBuilder cb =
            getEntityManager().getCriteriaBuilder();

        CriteriaQuery<ItemBidSummary> criteria =
            cb.createQuery(ItemBidSummary.class);

        Root<Item> i = criteria.from(Item.class);

        // Some query details...
        // ...

        TypedQuery<ItemBidSummary> query =
            page.createQuery(em, criteria, i);

        return query.getResultList();
    }

    // ...
}
```

① 这是你之前已经多次看到过的一个常规条件查询。

② 将结束查询委托给指定 Page。

具体的 Page 实现会准备该查询，设置必要的偏移量、限制以及搜寻参数。

实现偏移量分页

例如，以下是偏移量分页策略的实现：

路径: /apps/app-web/src/main/java/org/jpwh/web/dao/OffsetPage.java

```
public class OffsetPage extends Page {

    protected int current = 1;

    // ...

    @Override
    public <T> TypedQuery<T> createQuery(EntityManager em,
        CriteriaQuery<T> criteriaQuery,
        Path attributePath) {

        throwIfNotApplicableFor(attributePath);

        CriteriaBuilder cb = em.getCriteriaBuilder();

        Path sortPath = attributePath.get(getSortAttribute());
```

② 尝试解析

排序属性

③ 添加 ORDER BY

```

criteriaQuery.orderBy(
    isSortedAscending() ? cb.asc(sortPath) : cb.desc(sortPath)
);

TypedQuery<T> query = em.createQuery(criteriaQuery);

query.setFirstResult(getRangeStartInteger()); ← ④ 设置偏移量

if (getSize() != -1) ← ⑤ 设置结果大小
    query.setMaxResults(getSize());

return query;
}
}

```

- ❶ 对于基于偏移量的分页，需要知道你现在位于哪一页。默认情况下，从第 1 页开始。
- ❷ 验证这个页面的排序属性是否可以根据属性路径来解析，并且由此该模型是否可以被查询所使用。如果页面的排序属性在查询中引用的模型类上不可用，那么该方法就会抛出一个异常。这是一个安全机制，如果将错误的分页设置与错误的查询配对，该机制就会产生一条有意义的错误消息。
- ❸ 将 ORDER BY 子句添加到查询。
- ❹ 设置查询的偏移量：起始结果行。
- ❺ 使用期望的页面大小截取结果。

我们没有显示这里涉及的所有方法，比如 `getRangeStartInteger()`，它可以计算基于页面大小必须为当前页面检索的第一行的序号。可以在源代码中找到其他简单方便的方法。

注意，结果的顺序可能不是确定的：如果根据商品名称升序排列，并且有几个商品具有相同的名称，那么数据库就会以 DBMS 创建者认为合适的任何顺序返回它们。应该根据唯一的键属性或者添加一个具有键属性的额外顺序标准来排序。尽管许多开发人员选择回避且忽略使用偏移量方法的确定性排序问题，但对于搜寻策略来说，可预测的记录顺序是强制性的。

实现搜寻分页

对于搜寻分页，需要将限制添加到一个查询。我们假设上一页显示了到 Coffee Machine 为止的所有按照名称升序排列的商品，如图 19-2 所示，并且你希望使用一个 SQL 查询来检索下一页。你记住了上一页的最后一个值、Coffee Machine 记录及其标识符值(比如 5)，所以可以在 SQL 中编写：

```

select i.* from ITEM i
where
    i.NAME >= 'Coffee Machine'
    and (
        i.NAME <> 'Coffee Machine'
        or i.ID > 5
    )
order by
    i.NAME asc, i.ID asc

```

该查询的第一个限制表明,“给我所有名称大于或等于[Coffee Machine]的商品,”它会向前搜寻到上一页的结尾处。数据库可以用索引扫描执行这个有效限制。然后要通过表明不想要名称为 Coffee Machine 的记录来进一步限制结果,这样就能跳过已经在上一页中显示过的最后一条记录。

不过数据库中可能存在两个名称为 Coffee Machine 的商品。要阻止商品落在页面之间的缝隙中,一个唯一键就是必不可少的。必须使用该唯一键对结果进行排序并且限制结果。这里使用了主键,因此会确保数据库只包括名称不是 Coffee Machine 的商品,或者(即便名称为 Coffee Machine)使用一个比你在上一页中显示的标识符值更大的标识符值。

当然,如果商品名称(或者排序依据的任何其他列)是唯一的,那么就不需要一个额外的唯一键。该通用示例代码假设你总是会提供一个显式的唯一键属性。还要注意,商品标识符实际上是递增的数值并不重要;重要的方面是这个键允许确定的排序。

可以使用行值构造函数语法在 SQL 中以更紧凑的形式编写这个相同查询:

```
select i.* from ITEM i
where
    (i.NAME, i.ID) > ('Coffee Machine', 5)
order by
    i.NAME asc, i.ID asc
```

这种限制表达式甚至适用于 Hibernate 中的 JPQL。不过, JPA 没有标准化它;它不可用于条件 API 并且不被所有的数据库产品支持。我们更愿意使用更冗长一些的变体,它适用于所有地方。如果按降序排列,那么就要将比较从大于反转为小于。

SeekPage 实现中的以下代码将这样一个限制添加到了一个条件查询:

路径: /apps/app-web/src/main/java/org/jpwh/web/dao/SeekPage.java

```
public class SeekPage extends Page {

    protected SingularAttribute uniqueAttribute;
    protected Comparable lastValue;
    protected Comparable lastUniqueValue;

    // ...

    @Override
    public <T> TypedQuery<T> createQuery(EntityManager em,
                                         CriteriaQuery<T> criteriaQuery,
                                         Path attributePath) {
        throwIfNotApplicableFor(attributePath);

        CriteriaBuilder cb = em.getCriteriaBuilder();

        Path sortPath = attributePath.get(getSortAttribute());
        Path uniqueSortPath = attributePath.get(getUniqueAttribute());
        if (isSortedAscending()) {
            criteriaQuery.orderBy(cb.asc(sortPath), cb.asc(uniqueSortPath));
        } else {
            criteriaQuery.orderBy(cb.desc(sortPath),
```

① 添加分页属性

② 记住上一页的值

③ 对结果排序


```

        cb.desc(uniqueSortPath));
    }
    applySeekRestriction(em, criteriaQuery, attributePath);
④添加限制 TypedQuery<T> query = em.createQuery(criteriaQuery);
    if (getSize() != -1) ← ⑤设置结果大小
        query.setMaxResults(getSize());
    return query;
}
// ...
}

```

- ❶ 除了这个常规排序属性，搜寻技术还需要一个分页属性，该属性必须是一个受保障的唯一键。这可以是实体模型的任何唯一属性，但它通常是主键属性。
- ❷ 对于排序属性和唯一键属性这两者来说，你都必须记住“上一页”中它们的值。然后可以通过搜寻那些值来检索下一页。任何可比较的值都行，这是条件查询中该限制 API 所要求的。
- ❸ 必须总是根据排序属性和唯一键属性这两者来对结果排序。
- ❹ 将所有必要的额外限制(未显示的)添加到该查询的 where 子句，以便为目标页面搜寻超出最后一个已知值的值。
- ❺ 使用期望的页面大小截取结果。

可以在示例代码中找到完整的 `applySeekRestriction()` 方法；这是条件查询代码，我们没有足够的空间在这里显示它。最后一个查询等同于你之前看到过的 SQL 示例。

我们来测试持久化层的新分页功能。

为搜寻方法找到页面边界

我们之前说过，使用搜寻技术直接跳转到一个页面并不容易，因为给定特定的页数，也不知道要搜寻的最后值是什么。可以使用一个像这样的 SQL 语句来弄明白那些值是什么：

```

select i.NAME, i.ID
from ITEM i
where
    (select count(i2.*)
     from ITEM i2
     where (i2.NAME, i2.ID) <= (i.NAME, i.ID)
    ) % :pageSizeParameter = 0
order by i.NAME asc, i.ID asc

```

这个带有一个模运算(%)的查询会返回所有的(NAME, ID)配对，它们都是页面边界值：它们是每个页面上最后的值。我们已经在示例代码中包含了这个查询的条件版本。

19.2.3 逐页查询

现在当调用 `ItemDAO#getItemBidSummaries()` 方法时，必须提供一个 Page 实例。持久

化层之上的服务或 UI 层客户端会执行以下代码：

路径：/apps/app-web/src/test/java/org/jpwh/test/service/PagingTest.java

页面大小

```
OffsetPage page = new OffsetPage(
    3,
    itemDAO.getCount(),
    Item_.name, ASC,
    Item_.name, Item_.auctionEnd, Item_.maxBidAmount
);
```

← 可用的总记录数

← 默认排序属性和排序方向

← 所有允许作为这个页面排序属性的属性

```
List<ItemBidSummary> result = itemDAO.getItemBidSummaries(page);
```

前进到下一页很容易，并且可以直接跳转到任何页数：

```
page.setCurrent(2);
result = itemDAO.getItemBidSummaries(page);
```

搜寻技术需要更多一点信息。首先要用额外所需的唯一键属性实例化一个 SeekPage：

路径：/apps/app-web/src/test/java/org/jpwh/test/service/PagingTest.java

```
SeekPage page = new SeekPage(
    3,
    itemDAO.getCount(),
    Item_.name, ASC,
    Item_.id,
    Item_.name, Item_.auctionEnd, Item_.maxBidAmount
);
```

← 用于排序和搜寻的额外唯一键属性

```
List<ItemBidSummary> result = itemDAO.getItemBidSummaries(page);
```

使用偏移量策略，要跳转到一个页面你所必须知道的就是页数。

使用搜寻策略，必须记住上一页所显示的最后值：

```
ItemBidSummary lastShownOnPreviousPage = // ...
```

```
page.setLastValue(lastShownOnPreviousPage.getName());
```

```
page.setLastUniqueValue(lastShownOnPreviousPage.getItemId());
```

```
result = itemDAO.getItemBidSummaries(page);
```

显然这对于持久化层的一个客户端来说会有更多的工作量。相较于一个简单数字，它必须能够动态记住最后值，这取决于排序和唯一属性。

我们使用 JSF 和 CDI 为之前的截屏实现了 UI，包括用于我们的分页 API 的必要粘滞代码。看看源代码，以便获得如何将这一技术集成到你自己的服务和 UI 层中的灵感。

即使不使用 JSF，在其他 Web 框架中采用我们的分页和排序解决方案也应该很简单。如果计划使用 JSF，那么应该阅读下一节：我们将使用 JSF Web 接口和状态性服务层来深入研究复杂的用例，比如放置一个出价以及编辑一个拍卖商品。

19.3 构建 JSF 应用程序

在上一个示例中，可以在分类中单击一个拍卖商品的名称。这会将你带到该商品的拍卖页面，它具有更为详尽的信息，如图 19-3 所示。

商品拍卖页面有一个表单，允许你对该商品出价。这是要实现的第一个用例：放置一个出价。可以使用一个简单的请求作用域服务来处理它。

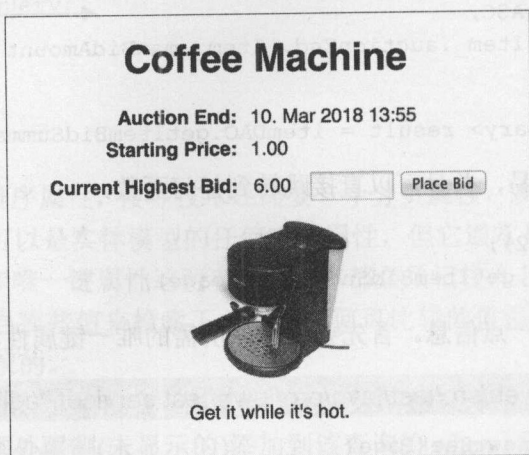


图 19-3 浏览商品详情并且放置一个出价

19.3.1 请求作用域服务

我们首先从用于页面的 XHTML 模板的一些重要摘录开始介绍：

路径：/apps/app-web/src/main/webapp/auction.xhtml

```
<f:metadata>
    <f:viewParam name="id" value="#{auctionService.id}"/>
</f:metadata>
```

该页面是一个书签性的——也就是说，它可以并且将会被一个查询参数调用，即 `Item` 的标识符值。该视图元数据会告知 JSF 它应该将这个参数值绑定到 `AuctionService` 的后端属性 `id`。你稍后要实现这个新服务。

要在该页面上呈现一些商品详情，这需要要求 `AuctionService` 提供数据：

路径：/apps/app-web/src/main/webapp/auction.xhtml

```
<h:outputText value="Auction End:"/>
<h:outputText value="#{auctionService.item.auctionEnd}">
    <f:convertDateTime pattern="dd. MMM yyyy HH:mm"/>
</h:outputText>

<h:outputText value="Starting Price:"/>
<h:outputText value="#{auctionService.item.initialPrice}"/>
```


呈现此信息会反复调用 `AuctionService#getItem()` 方法，当处理该服务实现时，必须牢记一些事情。最后，以下是放置一个出价的表单：

路径：/apps/app-web/src/main/webapp/auction.xhtml

```
<h:form>
  <h:inputHidden value="#{auctionService.id}"/>  ← ❶ 发送标识符
  <h:inputText value="#{auctionService.newBidAmount}"
    size="6"/>  ← ❷ 设置出价金额
  <h:commandButton value="Place Bid"
    action="#{auctionService.placeBid}"/>
</h:form>  ← ❸ 放置出价
```

- ❶ 需要在提交表单时传输该商品的标识符值。后端服务是请求作用域的，因此需要为每个请求初始化它：这会调用 `AuctionService#setId()` 方法。
- ❷ 所输入的出价金额是在这个表单的 POSTback 被处理时使用 `AuctionService#setNewBidAmount()` 方法由 JSF 设置的。
- ❸ 在所有的值都已经被绑定后，就会调用活动方法 `AuctionService#placeBid()`。

我们现在继续介绍后端服务类，它是一个简单的 CDI 托管 bean：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/AuctionService.java

```
@Named
@RequestScoped  ← ❶ 无须持有状态
public class AuctionService {

    @Inject
    ItemDAO itemDAO;

    @Inject
    BidDAO bidDAO;
    long id;  ← ❷ 定义状态
    Item item;
    BigDecimal highestBidAmount;
    BigDecimal newBidAmount;

    // ...
}
```

- ❶ 无须在这个用例的请求期间持有状态。在用一个 GET 请求呈现拍卖页面视图时，会创建一个服务实例，并且 JSF 会通过调用 `setId()` 来绑定请求参数。该服务实例会在呈现完成之后被销毁。服务器不会持有请求之间的任何状态。当拍卖表单被提交并且 POST 请求的处理开始时，JSF 会调用 `setId()` 来绑定隐藏的表单字段，可以再次初始化该服务的状态。
 - ❷ 为每个请求持有的状态就是用户当前正在处理的 Item 的标识符值、被加载之后的 Item、该商品的当前最高出价金额以及用户输入的新出价金额。
- 记住，在 JSF 中，任何在 XHTML 模板中绑定的属性访问器方法都可能被多次调用。

在 `AuctionService` 中，当调用访问器方法时，就会加载数据，并且必须确保你不会意外地从数据库中多次加载：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/AuctionService.java

```
public class AuctionService {

    public void setId(long id) {
        this.id = id;
        if (item == null) {
            item = itemDAO.findById(id);
            if (item == null)
                throw new EntityNotFoundException();
            highestBidAmount = itemDAO.getMaxBidAmount(item);
        }

        // Other plain getters and setters...
    }
}
```

当 JSF 在一个请求中首次调用 `setId()` 时，要用该标识符值加载 `Item` 一次。如果没有找到该实体实例，则会立即失败。还要在视图可以被呈现或者拍卖方法调用之前加载当前最高出价金额以便完全初始化这个服务的状态。

注意，这是一个非事务性方法！不同于 EJB，一个简单 CDI bean 中的方法无须默认激活一个事务。由 DAO 所使用的已生成的 `EntityManager` 和持久化上下文是非同步的，而你要在自动提交模式中从数据库读取数据。即便是之后，当仍旧在处理相同请求时，一个事务性方法会被调用，也还是会重用相同的非同步、已经生成的、请求作用域的 `EntityManager`。

`placeBid()` 方法就是这样一个事务性方法：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/AuctionService.java

```
public class AuctionService {
    // ...

    @Transactional
    public String placeBid() {
        itemDAO.joinTransaction();

        if (!getItem().isValidBidAmount(
            getHighestBidAmount(),
            getNewBidAmount()
        )) {
            ValidationMessages.addFacesMessage("Auction.bid.TooLow");
            return null;
        }

        itemDAO.checkVersion(getItem(), true);

        bidDAO.makePersistent(new Bid(getNewBidAmount(), getItem()));

        return "auction?id=" + getId() + "&faces-redirect=true";
    }
}
```

① 拦截器会包装事务上下文中的方法调用

② 存储出价

③ 检查较高的出价

④ 强制版本递增

⑤ 重定向

- ❶ `@Transactional` 注解是 Java EE 7 中的一个新注解(来自 JTA 1.2), 并且类似于 EJB 组件上的 `@TransactionAttribute`。在内部, 一个拦截器会包装系统事务上下文中的方法调用, 就像 EJB 方法一样。
- ❷ 执行事务性工作并在数据库中存储一个新出价, 防止并发出价。必须用事务联结持久化上下文。调用哪个 DAO 无关紧要: 它们全部都共享相同的请求作用域的 `EntityManager`。
- ❸ 如果在用户思考并且查看所呈现的拍卖页面时, 提交了另一个较高出价的事务, 则会用一条消息提示用户出价失败以及重新呈现拍卖页面。
- ❹ 必须在刷新时强制 Item 版本递增, 以避免并发出价。如果在此期间运行了另一个事务, 并且在 `setId()` 中从数据库加载了相同 Item 版本以及当前最高出价, 那么在 `placeBid()` 中有一个事务就必须失败。
- ❺ 这在 JSF 中是一个简单的 POST 后的重定向, 因此用户可以在提交一次出价之后安全地重新加载该页面。

当 `placeBid()` 方法返回时, 该事务就会被提交, 而联结的持久化上下文会自动将变更刷新到数据库。在 Java EE 7 中, 你最终会拥有 EJB 最便利的功能: 声明式事务分界, 独立于 EJB 之外。

Java EE 中的另一个新功能是 CDI 会话作用域及其在 JSF 中的集成。

19.3.2 会话作用域服务

可以将 Java EE 中的 beans 以及生成的实例声明为 `@ConversationScoped`。这一特殊作用域是使用标准 `javax.enterprise.context.Conversation` API 来实现手动控制的。要理解会话作用域, 思考一下上一章中的示例。

最简短的可能会话, 从应用程序用户角度来看的工作单元, 就是单个请求/响应循环。一个用户发送一个请求, 而服务器开启一个会话上下文来为该请求保留数据。当响应被返回时, 这个短期运行的瞬时会话上下文就会结束并且被关闭。因此瞬时会话作用域就与请求作用域相同; 这两个上下文具有相同的生命周期。这就是 CDI 规范将会话作用域映射到 `servlet` 请求的方式, 因而也就是映射到 JSF 和 JAX-RS 请求的方式。

使用 `Conversation` API, 你就可以在服务器上提升一个会话并且让它长期运行而不再是瞬时的。如果在一个请求期间提升会话, 那么相同的会话上下文就可用于下一个请求。可以使用该上下文来将数据从一个请求传递到下一个请求。通常为此你会需要会话控制上下文, 但来自几个并行会话的数据(一个用户打开了两个浏览器选项卡)在会话控制中必须被手动隔离。这就是会话上下文所提供的东西: 可控制上下文中的自动化数据隔离, 这比普通的会话控制上下文更加方便。

当然, 你要使用服务器端的会话控制实现会话上下文, 并且存储用户会话控制中的数据。为了隔离和区分一个会话控制中的多个会话, 每个会话都要具有一个标识符值: 必须将该值与每一个请求一起传递。为此, 甚至已经标准化了参数名称 `cid`。每个会话还要有一个单独的超时设置, 以便服务器可以在用户停止发送请求时释放资源。这样就使得服务器可以自动清理过期的会话状态, 而无须等待整个用户会话过期。

长期运行(非瞬时)的会话上下文的另一个很好的功能是,服务器会自动保护对会话作用域数据的并发访问:如果一个用户多次快速单击一个操作按钮,并且每个请求都传递特定会话的标识符值,那么服务器将使用 `BusyConversationException` 终止除了其中一个请求之外的所有请求。

常见问题解答: JSF 流作用域又如何呢?

在最新的 JSF 版本中添加的流作用域类似于 CDI 会话作用域。它专注于如何分组和发现工作流的视图模板,以及工作流中的导航规则如何才能变成自动化的。也可以使用它来跨请求持有状态。遗憾的是,会话中的流上下文超时并未指定,也没有手动结束或者提升工作流上下文的一个控制 API。我们认为当前版本的流作用域在大多数简单用例之外并不可用——只能用于服务器上大量资源消耗无关紧要的场景。我们希望 Java EE 的未来版本会与 CDI 会话以及 JSF 流作用域功能保持一致,并且合并这两者的优势。

下面介绍 JSF 中会话作用域使用的一个示例。你要实现一个使用工作流的应用,这需要用户的几个请求来完成:将一个商品上架拍卖。

“编辑商品”会话工作流

如果细细想一想它,创建和编辑一个拍卖商品是非常类似的任务。这些工作流都是会话:从应用程序用户角度来看的工作单元,而你的用户将期望它们看起来是类似的。因此你的目标是避免代码重复;应该使用单个 UI 和后端服务来实现这两个用例。应用程序会通过工作流指引用户;图 19-4 显示了一个作为状态图的图形表示。

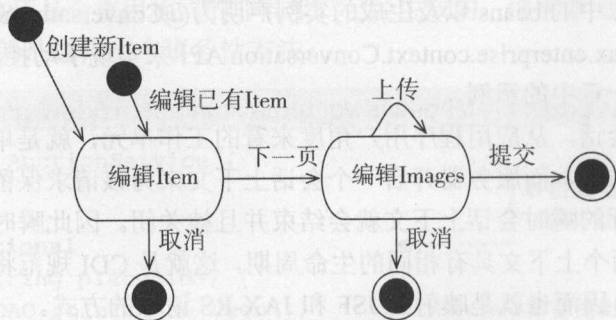


图 19-4 “编辑商品”会话的工作流

这里是如何读取这个图表的方式。如果没有商品存在的话,用户可以不使用任何数据来开启会话。或者,用户可以使用一个已有商品的标识符值,它可能是一个简单数值。用户如何获得这个标识符并不重要;可能他们在上一个会话中执行了某种搜索。这是一个常见场景,并且许多会话工作流都有几个入口点(该状态图中的实心圆)。

从用户的角度来看,编辑一个商品是一个多步骤的处理过程:每一个圆角框都表示应用程序的一个状态。在编辑商品状态,应用程序会为用户呈现一个对话框或表单,用户可以在其中修改商品的描述或者起拍价。同时,应用程序会等待用户触发一个事件;我们称之为用户思考时间。

当用户触发下一个事件时,应用程序就会处理它,而用户将进入下一个(等待)状态,编辑图片。在用户完成商品图片的处理时,他们会通过提交该商品和图片(具有一个外圈的

实心圆)来结束会话。单击任何页面上的取消按钮会终止所有工作。从应用程序用户的角度来看,整个会话就是一个原子单元:当用户单击提交按钮时,所有的变更都会被提交和完成。应该没有其他的转换或输出引发对数据库的任何持续性变更。

你要使用具有两个网页的向导风格用户界面来实现这一点。在第一个页面上,用户要输入商品的详情以便售卖或编辑,如图 19-5 所示。

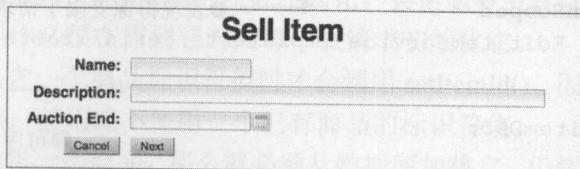


图 19-5 商品拍卖向导中的第一个页面

注意,呈现此页面不会在服务器上开启一个长期运行的会话上下文。这样会浪费资源,因为你尚不知道用户是否希望继续进行以及处理商品详情。在用户首次单击下一页按钮时,你就开启了长期运行的会话上下文并且在服务器上跨请求持有了状态。如果表单验证通过,那么服务器就会存储用户会话控制中的会话性数据并且将用户推进到编辑图片页面(参见图 19-6)。



图 19-6 在向导的第二个页面上编辑图片

当用户在上传了该商品图片之后提交商品时,工作单元就完成了。用户可能随时单击取消按钮来结束会话或者离开终端让会话过期。用户可能会在两个浏览器选项卡中进行处理并且并行运行几个会话,因此在服务器上的相同用户会话控制中必须对它们彼此隔离。如果用户会话控制过期(或者,如果在使用粘滞会话控制的服务器群集中,一台服务器出现故障),那么这个用户的所有会话数据就会被移除。

用于此向导的 XHTML 模板没有用于该会话的标记;JSF 与 CDI 一起会自动化处理它。如果在表单被呈现时发现了一个长期运行的会话上下文,那么会话标识符就会在所生成的隐藏字段中与常规 JSF 表单提交一起被自动传递。这甚至能跨 JSF 重定向来工作:cid 参数被自动附加到该重定向目标 URL。

绑定到那些页面的会话作用域后端服务——EditItemService——是所有处理运行的地方,也是你控制上下文的地方。

一个会话中的持久化上下文

EditItemService 是整个向导和工作流的后端：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```

@Named
@ConversationScoped
public class EditItemService implements Serializable {

    @Inject
    ItemDAO itemDAO;

    @Inject
    ImageDAO imageDAO;

    // ...

    @Inject
    Conversation conversation;

    Long itemId;
    Item item = new Item();

    transient Part imageUploadPart;

    public void setItemId(Long itemId) {
        this.itemId = itemId;
        if (item.getId() == null && itemId != null) {
            item = itemDAO.findById(itemId);
            if (item == null)
                throw new EntityNotFoundException();
        }
    }

    // ...
}

```

① 表现得像受限于请求作用域
 ② 必须是可序列化的
 ③ 可序列化
 ④ 调用会话 API
 ⑤ 服务状态
 ⑥ 服务瞬时状态
 ⑦ 加载商品
 ⑧ 从数据库得到商品

- ① 该服务实例受限于会话作用域。默认情况下，会话上下文是瞬时的，因此它表现得像一个请求作用域的服务。
- ② 这个类必须是可序列化的，不同于请求作用域实现。EditItemService 的一个实例可以被存储在 HTTP 会话中，并且该会话控制数据可以被序列化到硬盘或者跨群集中的网络来发送。我们在第 18 章中使用了状态性 EJB 这一简单方式，也就是说，“它是不可钝化的。” CDI 会话作用域中的所有内容都必须是可钝化的，并且因而也就是可序列化的。
- ③ 注入的 DAO 实例具有独立的作用域并且可被序列化。可以认为它们不能被序列化，因为它们有一个 EntityManager 字段，这是无法序列化的。我们稍后将探讨此不匹配。
- ④ Conversation API 是由容器提供的。调用它来控制会话上下文。在用户首次单击下一页按钮时，需要它来将瞬时会话提升为长期运行会话。

- ⑤ 这是服务的状态：用户正在向导页面上编辑的商品。首先要处理瞬时状态中的一个新 `Item` 实体实例。如果这个服务是使用一个商品标识符值来初始化的，就要在 `setItemId()` 中加载该 `Item`。
- ⑥ 这是服务的一个瞬时状态。你只会在用户单击编辑图片页面上的上传按钮时才临时需要它。`Servlet API` 的 `Part` 类不能被序列化。会话服务中具有一些瞬时状态的情况并非罕见，但你必须在需要它时为每一次请求初始化它。
- ⑦ 只有在该请求包含一个商品标识符值时才会调用 `setItemId()`。因此要在这个会话中使用两个入口点：使用或不使用一个已有商品的标识符值。
- ⑧ 如果用户正在编辑一个商品，那么就必須从数据库加载它。仍然要依赖一个请求作用域的持久化上下文，因此一旦请求完成，这个 `Item` 实例就会处于分离状态。可以在会话服务的状态中保留分离的实体实例，并且在需要持久化变更时合并它(参阅 10.3.4 节)。

不同于状态性 EJB，你无法禁用会话上下文的钝化。CDI 规范要求，类和会话作用域的组件的所有非瞬时依赖都要可序列化。实际上，如果犯了一个错误并且包含了无法在会话作用域的 `bean` 中被序列化的状态或者任何其依赖项，你将会得到一个部署错误。

为了通过此测试，我们之前所说的 `GenericDAO` 是 `java.io.Serializable` 实际上隐瞒了这一情况。`GenericDAOImpl` 中的 `EntityManager` 字段是不可序列化的！这是可行的，因为 CDI 使用了上下文相关的引用——智能占位符。

DAO 的 `EntityManager` 字段在运行时不是持久化上下文的一个真实实例。该字段会持有对一些 `EntityManager` 的引用：当前的一些持久化上下文。记住，CDI 会通过构造函数生成并且注入依赖项。因为你将它声明成了请求作用域的，所以在运行时注入一个仅仅看起来像真实 `EntityManager` 的特殊代理。这个代理会将所有的调用委托给它在当前请求上下文中找到的一个真实的 `EntityManager`。该代理是可序列化的并且不会在请求完成之后持有对 `EntityManager` 的引用。然后它可以很容易地被序列化；并且当它被反序列化时，甚至可能是在不同的 JVM 上，它将继续执行其工作并且在无论何时被调用时获得一个请求作用域的 `EntityManager`。因此，你没有序列化整个持久化上下文——而是仅仅序列化了一个代理，它能查找当前请求作用域的持久化上下文。

这在一开始听起来可能很奇怪，但它就是 CDI 的工作方式：如果一个请求作用域的 `bean` 被注入到一个会话作用域、会话控制作用域或应用程序作用域的 `bean` 中，就需要一个间接引用。如果在没有活动的请求上下文(比如在一个 `servlet` 的 `init()` 方法中)时尝试通过 DAO 调用 `EntityManager` 代理，就会得到一个 `ContextNotActiveException`，因为该代理无法获得一个当前的 `EntityManager`。CDI 规范也做了定义，就算在代理表示的组件可能无法被钝化时，这样的代理也可以被钝化(序列化)。

现在假设用户已经用商品详情填充了向导第一页上的表单，并且单击下一页按钮。必须提升该瞬时会话上下文。

开启长期运行的会话

在向导中单击下一页按钮会提交具有 `Item` 详情的表单，并且会将用户带到“编辑图片”页面。因为 `EditItemService` 被绑定到了瞬时会话上下文，所以会在一个新的、瞬时会话上

下文中处理这个请求。记住，该瞬时会话作用域与请求作用域相同。

当使用一个操作方法处理该请求时，就要调用 Conversation API 来控制当前的瞬时会话上下文：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
public class EditItemService implements Serializable {
    // ...

    public String editImages() {
        if (conversation.isTransient()) {
            conversation.setTimeout(10 * 60 * 1000); ← 10 分钟
            conversation.begin();
        }
        return "editItemImages";
    }
    // ...
}
```

在 JSF 引擎已经将 Item 的所有详情设置到 EditItemService#item 属性之后，就会调用该操作方法。要使用一个单独的超时设置让该瞬时会话长期运行。这个超时明显要小于或等于用户会话的超时；较大的值并没有什么意义。服务器会保留用户会话中服务的状态，并且在“编辑图片”页面上将一个自动生成的会话标识符呈现为任意操作表单上的一个隐藏字段。如果需要它，也可以使用 Conversation#getId() 获得该会话的标识符值。甚至可以在 Conversation#begin() 方法调用中设置你自己的标识符值。

服务器现在会等待具有该会话标识符的下一个请求，很可能是来自“编辑图片”页面。如果等待时间过长，可能是由于长期运行的会话或者整个会话超时，则会根据请求抛出一个 NonexistentConversationException 异常。

如果用户希望将一张图片附加到拍卖商品，那么下一个请求可能就是绑定到这个服务方法的一个上传操作：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
public class EditItemService implements Serializable {
    // ...

    public void uploadImage() throws Exception {
        if (imageUploadPart == null)
            return;

        Image image =
            imageDAO.hydrateImage(imageUploadPart.getInputStream());
        image.setName(imageUploadPart.getSubmittedFileName());
        image.setContentType(imageUploadPart.getContentType());

        image.setItem(item);
        item.getImages().add(image);
    }
}
```

① 创建实例

② 添加瞬时 Image

```

    }
    // ...
}

```

① 从提交的多部分表单创建 **Image** 实体实例。

② 必须将瞬时 **Image** 添加到瞬时 **Item** 或分离 **Item**。当上传的图片数据被添加到会话状态继而添加到用户会话时，这个会话就会消耗服务器上越来越多的内存。

在构建一个状态性服务器系统时，一个最重要的问题就是内存消耗以及系统如何才能处理许多并发的用户会话。必须小心使用会话数据。你是否必须为整个会话保留从用户处得到的数据或者从数据库中加载的数据。

当用户单击“提交商品”按钮时，该会话就结束了，并且所有的瞬时和分离实体实例必须被存储。

结束长期运行的会话

会话工作流程会在瞬时或分离 **Item** 及其图片被存储时结束：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```

public class EditItemService implements Serializable {
    // ...

    @Transactional
    public String submitItem() {
        itemDAO.joinTransaction();
        // ...

        item = itemDAO.makePersistent(item);
        if (!conversation.isTransient())
            conversation.end();

        return "auction?id=" + item.getId() + "&faces-redirect=true";
    }
    // ...
}

```

① 包装方法调用

② 将持久化上下文与事务联结起来

③ 让 **Item** 持久化

④ 结束会话

⑤ 重定向

① 系统事务拦截器会包装方法调用。

② 如果希望存储数据，就必须将非同步的请求作用域的持久化上下文与系统事务联结起来。

③ 这个 DAO 调用会让瞬时或分离 **Item** 持久化。因为你在使用 **@OneToMany** 上的一个级联规则启用了它，所以它也会存储所有新的瞬时或者旧的分离 **Item#images** 集合元素。根据 DAO 契约，你必须将返回的实例用作当前状态。

④ 手动结束长期运行的会话。这实际上是降级：长期运行的会话变成瞬时的。你要在请求完成时销毁会话上下文以及这个服务实例。所有的会话状态都会从用户会话中移除。

- ⑤ 这在 JSF 中是使用当前持久化 Item 的新标识符值回到拍卖商品详情页面的 POST 之后的重定向。

另外，用户可以随时单击取消按钮来退出向导：

路径：/apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
public class EditItemService implements Serializable {  
    // ...  
    public String cancel() {  
        if (!conversation.isTransient())  
            conversation.end();  
        return "catalog?faces-redirect=true";  
    }  
    // ...  
}
```

这样就完成了我们使用 JSF、CDI 和 JPA 持久化层的状态性服务的示例。我们认为将 JSF 和 CDI 与 JPA 结合使用是非常棒的；无论是从资源消耗还是代码量来说，你都会以很小的代价得到经过良好测试并且标准化的编程模型。

现在我们要继续使用 CDI，但相较于 JSF，我们要在为所有富客户端设计的无状态服务器中引入 JAX-RS 与 JPA 结合使用。在这个架构中，你将面临的其中一个挑战是序列化数据。

19.4 序列化域模型数据

当我们在 3.2.3 节中首次探讨编写具有持久化能力的类时，我们就简要介绍过，这些类不必实现 `java.io.Serializable`。可以在需要时应用这个标记接口。

到目前为止，其中一个必须使用它的情况出现在第 18 章中。域模型实例会在 EJB 客户端和服务系统之间被传递，并且它们会在一端被自动序列化成一些连线传输格式，以及在另一端被反序列化。这将完美运行，并且没有使用自定义，因为客户端和服务都是 Java 虚拟机，而 Hibernate 库可同时用于这两个系统上。客户端使用了远程方法调用(RMI)以及标准的 Java 序列化格式(表示 Java 对象的字节流)。

如果客户端没有运行在 Java 虚拟机中，那么你可能不希望从服务器接收表示 Java 对象的字节流。一个常见的场景是，处理一个富客户端的无状态服务器系统，比如运行在一个 Web 浏览器中的 JavaScript 应用程序或者一个移动设备应用程序。为了实现它，你通常要在服务器上创建一个使用 HTTP 进行通信的 Web API，并且传输 XML 或者 JSON 有效负荷，之后客户端必须解析它们。客户端还要在必须将变更存储在服务器上时发送 XML 或 JSON 数据，所以你的服务器必须能够生成和消费期望的媒体类型。

设计 RESTful 超媒体驱动的应用程序

许多人都将具有 HTTP 远程通信和 JSON 或 XML 媒体类型的系统称为 RESTful。但具

有具象状态传输的架构的其中一个最重要的方面是，客户端和服务端交换超媒体文档。这些超媒体文档包含数据以及数据上可用的可见功能(操作)：HATEOAS(Hypermedia As The Engine of Application State，超媒体即应用程序状态引擎)这个名称也由此而来。由于本书主要介绍 Hibernate 而非 Web API 设计，所以我们只能向你介绍如何构建一个交换基本 XML 文档的简单 HTTP API，它并非 RESTful 并且不使用超媒体。

当设计你自己的 API 时，请思考你选择的媒体类型的 H 因子(参见 <http://amundsen.com/hypermedia/hfactor/>)，并且研究一下由 Leonard Richardson、Mike Amundsen 和 Sam Ruby 编写的 *RESTful Web APIs* 这本非常好的书(Richardson 出版社，2013 年出版)。建议避免使用 JSON，因为它需要专有的扩展来改进其 H 因子。

设计你自己的基于 XML 的超媒体格式，也许是通过扩展本章中介绍的实例来实现，这好不了多少。我们偏爱的媒体类型是普通 XHTML：它有一个很大的 H 因子，并且使用随处可见的 API 就能很容易地编写和读取。经过压缩后，它会比 JSON 更为高效，并且在构建和测试你的 API 时与其交互会很愉悦。Jon Moore 在“使用 HTML 构建超媒体 API”一文中介绍了这样一个设计的极佳示例(www.infoq.com/presentations/web-api-html)。

你现在要使用 JAX-RS 框架编写一个 HTTP 服务器，生成和消费 XML 文档。尽管示例都是 XML 的，但它们同样适用于 JSON，并且我们所探讨的基本问题在这两者中都是相同的。

19.4.1 编写一个 JAX-RS 服务

我们首先处理 JAX-RS 服务。在客户端发送一个 GET HTTP 请求时，一个服务方法会传递一个表示 Item 实体实例的 XML 文档。另一个服务方法会接受 XML 文档，用于在 PUT 请求中更新一个 Item：

路径：`/apps/app-web/src/main/java/org/jpwh/web/jaxrs/ItemService.java`

```
@Path("/item")
public class ItemService {

    @Inject
    ItemDAO itemDAO;

    @GET
    @Path("/{id}")
    @Produces(APPLICATION_XML)
    public Item get(@PathParam("id") Long id) {
        Item item = itemDAO.findById(id);
        if (item == null)
            throw new WebApplicationException(NOT_FOUND);
        return item;
    }

    @PUT
    @Path("/{id}")
```

① 请求路径

② GET 请求

③ 调用的参数

④ 将值序列化到 XML

```

@Consumes (APPLICATION_XML)           ← ⑤ 反序列化 XML
@Transactional                         ← ⑥ 开启事务
public void put (@PathParam("id") Long id, Item item) {
    itemDAO.joinTransaction();
    itemDAO.makePersistent(item);
}
// ...
}

```

- ① 当服务器接收到具有请求路径/item 的请求时，这个服务上的方法会处理它。默认情况下，该服务实例是请求作用域的，但可以应用 CDI 作用域的注解来修改它。
- ② 一个 HTTP GET 请求会映射到这个方法。
- ③ 该容器会使用/item 之后的路径片段作为用于调用的参数值，比如/item/123。要使用 @PathParam 将它映射到一个方法参数。
- ④ 这个方法会生成 XML 媒体；因此，必须要将该方法的返回值序列化成 XML。当心：这个注解与 CDI 中的生产者注解不同。它是一个不同的包！
- ⑤ 这个方法会消费 XML 媒体；因此，必须对该 XML 文档反序列化并且将它转换成一个分离的 Item 实例。
- ⑥ 希望在这个方法中存储数据，因此必须开启一个系统事务并且将它与持久化上下文联结起来。

JAX-RS 标准涵盖了用于最重要媒体类型以及全部 Java 类型的自动封送处理。例如，JAX-RS 实现必须能够为应用程序提供的用于 XML 绑定的 Java 架构(Java Architecture for XML Binding, JAXB)类生产和消费 XML 媒体。因此，该域模型实体类 Item 必须变成一个 JAXB 类。

19.4.2 应用 JAXB 映射

JAXB 与 JPA 非常像，它会使用注解来声明一个类的功能。这些注解会将一个类的属性映射到 XML 文档中的元素和属性。JAXB 运行时会自动从 XML 文档中读取实例以及将实例写入 XML 文档。现在这对你来说听起来应该很熟悉；JAXB 对于启用了 JPA 的域模型来说是一个非常好的伙伴。

以下是一个样本 Item 的 XML 文档：

```

<item id="1" auctionEnd="2018-03-06T15:00:00+01:00">
  <name>Baseball Glove</name>
  <description>It is brown.</description>
  <initialPrice>5.00</initialPrice>
  <bids>
    <bid id="1" createdOn="2018-03-06T15:01:00+01:00">
      <amount>11.00</amount>
    </bid>
    <bid id="2" createdOn="2018-03-06T15:02:00+01:00">
      <amount>12.00</amount>
    </bid>
    <bid id="3" createdOn="2018-03-06T15:03:00+01:00">

```

卖家在哪里？稍后我们将探讨这个问题


```

        <amount>13.00</amount>
      </bid>
    </bids>
  </item>

```

制定了几个设计决策来提供这个 XML 结构。我们来看看 Item 类中的 JAXB 注解，以探究可用的选项：

路径: /apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```

@Entity
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Item implements Serializable {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    @XmlAttribute
    protected Long id;

    @NotNull
    @Future(message = "{Item.auctionEnd.Future}")
    @XmlAttribute
    protected Date auctionEnd;

    // ...
}

```

← ① 映射到 XML

← ② 调用字段

- ① 一个 Item 实例会映射到一个<item> XML 元素。这个注解实际上会在该类上启用 JAXB。
- ② 在序列化或反序列化一个实例时，JAXB 应该直接调用这些字段而非调用 getter 方法或 setter 方法。其背后的理由与 JPA 相同：方法设计中的自由度。

该商品的标识符和拍卖结束日期会变成 XML 属性，而其他所有属性都是嵌套的 XML 元素。你不必在 description 和 initialPrice 上放置任何 JAXB 注解；它们会默认映射到元素。域模型类的 Singular 属性很简单：它们要么是 XML 属性，要么是具有一些文本的嵌套 XML 元素。那么实体关联和集合又如何呢？

可以将一个集合直接嵌入到 XML 文档中，并且因而包含它，就像使用 Item#bids 所操作的那样：

路径: /apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```

public class Item implements Serializable {
    @OneToMany(mappedBy = "item")
    @XmlElementWrapper(name = "bids")
    @XmlElement(name = "bid")
    protected Set<Bid> bids = new HashSet<>();

    // ...
}

```

这里有一些可能的优化：当服务返回一个 `Item` 时，如果认为，“总是急加载，包括出价”，那么你就应该急加载它们。现在，JPA 需要几个查询加载 `Item` 以及默认延迟映射的 `Item#bids`。JAXB 序列化器会在准备好响应时自动遍历集合元素。

Hibernate 会延迟或急初始化集合数据，并且 JAXB(或你使用的任何其他序列化器)会依序列化每个元素。实际上 Hibernate 会在内部使用特殊的集合，正如 12.1.2 节中探讨过的，这在你序列化的时候没有任何区别。稍后当序列化一个 XML 文档时，它会很重要，但我们现在忽略这个问题。

当不希望 XML 文档中包含一个集合或者一个属性时，可以使用 `@XmlTransient` 注解：

路径：/apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```
public class Item implements Serializable {

    @OneToMany(mappedBy = "item", cascade = MERGE)
    @XmlTransient
    protected Set<Image> images = new HashSet<>();

    // ...
}
```

集合很容易被处理，无论它们是不是基元、可嵌入的或多值实体关联的集合。当然，必须小心使用循环引用，比如每个 `Bid` 使用一个对 `Item` 的(反向)引用。某些时候，必须打破它并且将一个引用声明为瞬时的。

在序列化由 Hibernate 加载的实体实例时，你面临的最难的问题就是内部代理：用于实体关联延迟加载的占位符。在 `Item` 类中，这就是 `seller` 属性，它引用了一个 `User` 实体。

19.4.3 序列化 Hibernate 代理

`Item#seller` 是使用 `@ManyToOne(fetch = LAZY)`——一个延迟实体关联——来映射的。当加载一个 `Item` 实体实例时，其 `seller` 属性并非一个真正的 `User`：它是一个 `User` 代理，来自 Hibernate 的一个运行时生成的类。

如果不声明其他任何东西，那么这就是 JAXB 将会呈现该属性的方式：

```
<item id="1" auctionEnd="2018-03-06T15:00:00+01:00">
  <!-- ... -->
  <seller/>
  <!-- ... -->
</item>
```

这样一个文档将向客户端表明，该商品没有卖家。这当然是错误的：一个未初始化的代理与 `null` 并不相同！可以将特殊的含义指定到一个空的 XML 元素并且表示，在客户端上，“一个空的元素意味着一个代理”并且“一个缺失的元素意味着 `null`。”遗憾的是，我们已经看到了序列化的解决方案，甚至是专为 Hibernate 设计的，这些解决方案不会造成这种差别。有些不是专为 Hibernate 设计的序列化解决方案，甚至可能会在它们发现一个 Hibernate 代理时立即失败和出现故障。

通常，必须自定义你的序列化工具来以某些有意义的方式处理 Hibernate 代理。在这个应用程序中，你想要为一个未初始化的实体代理使用以下 XML 数据：

```
<item id="1" auctionEnd="2018-03-06T15:00:00+01:00">
  <!-- ... -->
  <seller type="org.jpwh.web.model.User" id="123"/>
  <!-- ... -->
</item>
```

这是与代理相同的数据：由代理表示的实体类和标识符值。客户端现在知道这个商品确实有一个卖家并且存在该用户的标识符；如果需要，它可以请求这些数据。如果用户更新一个商品时你在服务器上接收到了这个 XML 文档，那么你就可以从实体类名称和标识符值重构一个代理。

应该编写一个模型类来表示这样一个实体引用，并且将它映射到 XML 元素和属性：

路径：/apps/app-web/src/main/java/org/jpwh/web/model/EntityReference.java

```
@XmlElement
@XmlAccessorType(XmlAccessType.FIELD)
public class EntityReference {

    @XmlAttribute
    public Class type;

    @XmlAttribute
    public Long id;

    public EntityReference() {
    }

    public EntityReference(Class type, Long id) {
        this.type = type;
        this.id = id;
    }
}
```

接下来，必须自定义对该 Item 的封送处理和解封处理，因此相较于一个真正的 User，EntityReference 会处理该 Item#seller 属性。在 JAXB 中，要在该属性上应用一个自定义类型适配器：

路径：/apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```
public class Item implements Serializable {

    @NotNull
    @ManyToOne(fetch = LAZY)
    @XmlJavaTypeAdapter(EntityReferenceAdapter.class)
    protected User seller;

    // ...
}
```


可以将 `EntityReferenceAdapter` 用于所有实体关联属性。它知道如何从 XML 读取一个 `EntityReference` 以及将 `EntityReference` 写入到 XML:

路径: `/apps/app-web/src/main/java/org/jpwh/web/jaxrs/EntityReferenceAdapter.java`

```
public class EntityReferenceAdapter
    extends XmlAdapter<EntityReference, Object> {
    EntityManager em;

    public EntityReferenceAdapter() { ← ❶ 写 EntityReference
    }
    public EntityReferenceAdapter(EntityManager em) { ← ❷ 读 EntityReference
        this.em = em;
    }

    @Override
    public Object marshal(Object entityInstance)
        throws Exception {

        Class type = getType(entityInstance);
        Long id = getId(type, entityInstance);
        return new EntityReference(type, id); ← ❸ 创建序列化表示形式
    }

    @Override
    public Object unmarshal(EntityReference entityReference)
        throws Exception {
        if (em == null)
            throw new IllegalStateException(
                "Call Unmarshaller#setAdapter() and " +
                "provide an EntityManager"
            );

        return em.getReference( ← ❹ 创建代理
            entityReference.type,
            entityReference.id
        );
    }
}
```

- ❶ JAXB 会在它生成一个 XML 文档时调用这个构造函数。在这种情况下，你不需要一个 `EntityManager`：该代理包含需要用来写一个 `EntityReference` 的所有信息。
- ❷ JAXB 必须在它读取一个 XML 文档时调用这个构造函数。需要一个 `EntityManager` 来从 `EntityReference` 得到一个 Hibernate 代理。
- ❸ 在写一个 XML 文档时，要使用 Hibernate 代理并且创建一个可序列化的表示。这会调用我们没有在这里显示的内部 Hibernate 方法。
- ❹ 在读取一个 XML 文档时，要使用序列化的表示并且创建一个附加到当前持久化上下文的 Hibernate 代理。

最后，当 XML 文档必须在服务器上被解封处理时，需要一个 JAX-RS 的扩展，它将使用当前请求作用域的 `EntityManager` 自动初始化这个适配器。可以在本书的示例代码中找

到这个 `EntityReferenceXMLReader` 扩展。

还有一些我们需要探讨的要点。首先，我们还没有谈到过解封处理集合。XML 文档中的所有 `<bids>` 元素都会在调用服务时被反序列化，并且会从这些数据中创建 `Bid` 的分离实例。可以在服务运行时在分离的 `Item#bids` 上访问它们。不过，不会或者不能再发生其他什么了：在 JAXB 解封处理期间创建的集合并非其中一个特殊的 Hibernate 集合。即使你已经在你的映射中启用了 `Item#bids` 集合的级联合并，它也会被 `EntityManager#merge()` 所忽略。

这类似于在上一节中解决了的代理问题。必须在某特定属性在 XML 文档中被解封处理时创建一个特殊 Hibernate 集合。必须调用一些 Hibernate 内部 API 来创建那个神奇的集合。我们建议你考虑将集合变成只读；一般来说，当将数据发送到客户端时，集合映射就是查询结果中嵌入数据的快捷方式。当客户端将一个 XML 文档发送到服务器时，它不应包含任何 `<bids>` 元素。在服务器上，只能在集合被合并之后在持久化 `Item` 上访问它(在合并期间忽略该集合)。

其次，你可能想知道我们的 JSON 示例在哪里。我们知道现在在你的应用程序中你最可能依赖 JSON，并且不依赖一个自定义 XML 媒体类型。JSON 是一个在 JavaScript 客户端中解析的便利格式。不好的消息是，不依赖专有框架的话，我们无法找到在 JAX-RS 中自定义 JSON 封送处理和解封处理的一种方式。尽管 JAX-RS 可以被标准化，它如何生成和读取 JSON 并非标准化的；有些 JAX-RS 实现使用了 Jackson，而其他的使用了 Jettison。还有用于 JSON 处理(JSON Processing, JSONP)的新的标准 Java API，一些 JAX-RS 实现可能在未来会依赖它。

如果希望将 JSON 与 Hibernate 一起使用，那么除了你喜欢的 JSON 封送处理工具之外，你必须编写我们为 JAXB 所编写的相同扩展代码。你必须自定义代理处理、代理数据如何被发送到客户端，以及如何使用 `em.getReference()` 将代理数据转回成实体引用。你肯定必须依赖框架的一些扩展 API，就像我们使用 JAXB 所做的那样，但相同的模式是适用的。

19.5 本章小结

- 介绍了在 Web 应用程序环境中集成 Hibernate 和 JPA 的许多方式。为 CDI 注入启用了 `EntityManager`，并且使用 CDI 和用于查找器查询的通用排序以及分页解决方案改进了持久化层。
- 介绍了 JSF Web 应用程序中的 JPA：如何使用一个 JPA 持久化上下文来编写请求作用域以及会话作用域的服务。
- 探讨了与实体数据序列化相关联的问题和解决方案，以及如何解决一个具有无状态客户端和 JAX-RS 服务器的环境中的那些问题。

20.1 大量和批量处理

首先我们看看 JPA 中如何标准化了的大批量语句，比如 `UPDATE` 和 `DELETE`，以及

扩展 Hibernate

20

第 20 章

本章内容简介:

- 执行大量和批量数据操作
- 使用共享缓存提升可扩展性

要使用对象/关系映射将数据移动到应用层之中,以便使用面向对象的编程语言来处理这些数据。在实现每个工作单元都涉及小到中等大小数据集的多个用户在线事务处理应用程序时,这是一个好的策略。

另一方面,需要大量数据的操作并不最适合于应用层。应该将该操作移动到更接近数据存放的地方。在一个 SQL 系统中,如果必须实现一个涉及数千行数据的操作,那么 DML 语句 UPDATE 和 DELETE 会直接在数据库中执行,并且这通常就足够了。更复杂的操作可能需要额外的程序,以便在数据库内部运行;因此,应该考虑将存储过程作为一种可行的策略。可以在 Hibernate 应用程序中随时回退到 JDBC 和 SQL。我们之前在第 17 章中探讨过这样一些选项。在本章中,我们将介绍如何避免回退到 JDBC 以及如何使用 Hibernate 和 JPA 执行大量和批量操作。

我们宣称使用对象/关系持久化层的应用程序优于使用直接 JDBC 构建的应用程序的一个主要理由就是缓存。尽管我们强烈主张大多数应用程序都应该良好设计,以便可以达到可接受的性能,而无需使用缓存,但毫无疑问对于某些类型的应用程序,尤其是对于以读取为主的应用程序或者在数据库中保留重要元数据的应用程序来说,缓存会对性能造成巨大的影响。此外,将高并发应用程序扩展到每秒数千个在线事务,这通常需要一些缓存来降低数据库服务器上的负荷。在探讨了大量和批量操作之后,我们要探究 Hibernate 的缓存系统。

JPA 2 的主要新功能

- 大量的会直接转换成 SQL UPDATE 和 DELETE 语句的更新和删除操作,现在在 JPQL、条件以及 SQL 执行接口中已经被标准化并且可用了。
- 为启用共享的实体数据缓存的配置设置和注解现在标准化了。

20.1 大量和批量处理

首先我们查看 JPQL 中的标准化了的大批量语句,比如 UPDATE 和 DELETE,以及与

它们等效的条件版本。在那之后，我们要使用 SQL 原生语句重复这样一些操作。然后，你要学习如何批量插入和更新大量实体实例。最后，我们会介绍特殊的 `org.hibernate.StatelessSession` API。

20.1.1 JPQL 和条件中的大批量语句

Java 持久化查询语言类似于 SQL。这两者之间的主要区别在于，JPQL 使用类名替代表名并且使用属性名替代列名。JPQL 也理解继承性——也就是说，你是否正在使用一个超类或接口进行查询。JPA 条件查询设施支持与 JPQL 相同的查询结构，但此外还提供了类型安全以及容易的程式化语句创建。

接下来我们要介绍的语句支持在数据库中直接更新和删除数据，而无须将它们检索到内存中。我们还提供了一个语句，它可以选择数据并且在数据库中将它作为新的实体实例直接插入。

更新和删除实体实例

JPA 提供了比普通 SQL 更强大一些的 DML 操作。我们来看看 JPQL 中的第一个操作：一个 UPDATE 操作。见代码清单 20.1。

代码清单 20.1 执行一个 JPQL UPDATE 语句

```
Query query = em.createQuery(
    "update Item i set i.active = true where i.seller = :s"
).setParameter("s", johndoe);

int updatedEntities = query.executeUpdate();
assertEquals(updatedEntities, 2);  ←—— 实体实例，而非“行”
```

此 JPQL 语句看起来就像一个 SQL 语句，但它使用了一个实体名(类名)以及几个属性名。别名是可选的，因此也可以写成 `update Item set active = true`。要使用标准的查询 API 来绑定命名和位置参数。该 `executeUpdate` 调用会返回更新的实体实例数量，它们可能与更新的数据库行数不一样，这取决于映射策略。

此 UPDATE 语句仅会影响数据库；Hibernate 不会更新任何你已经检索到持久化上下文中的 Item 实例。在前面几章中，我们已经反复强调过，你应该考虑实体实例的状态管理，而不是考虑是如何管理 SQL 语句的。这个策略会假设你正在引用的实体实例在内存中可用。如果直接在数据库中更新或删除数据，则已经加载到应用程序内存中、加载到持久化上下文中的内容并不会被更新或删除。

避免这个问题的一个实用解决方案是一个简单的约定：首先在一个新的持久化上下文中执行所有直接的 DML 操作。然后，使用 `EntityManager` 来加载和存储实体实例。这个约定会确保该持久化上下文不被任何之前执行的语句所影响。或者，如果知道实体实例的状态在持久化上下文之外已经被修改了的话，就可以选择性地使用 `refresh()` 操作来将该实体实例的状态从数据库重新加载到持久化上下文中。

大批量 JPQL/条件语句和二级缓存

在数据库上直接执行一个 DML 操作会自动清除可选的 Hibernate 二级缓存。Hibernate 会解析你的 JPQL 和条件大批量操作，并且检测影响了哪些缓存区域。然后 Hibernate 会清除二级缓存中的区域。注意，这是一个粗粒度的无效化：尽管你可能只更新或删除了 ITEM 表中的几行，但 Hibernate 也会清除并且无效化它保留 Item 数据的所有缓存区域。

这是使用条件 API 的相同操作：

```
CriteriaUpdate<Item> update =
    criteriaBuilder.createCriteriaUpdate(Item.class);
Root<Item> i = update.from(Item.class);
update.set(i.get(Item_.active), true);
update.where(
    criteriaBuilder.equal(i.get(Item_.seller), johndoe)
);

int updatedEntities = em.createQuery(update).executeUpdate();
```

另一个好处是，JPQL UPDATE 语句和 CriteriaUpdate 会处理继承层次结构。如果所有者的姓名以“J”开头，那么以下语句就会将所有的信用卡标记为失窃：

```
Query query = em.createQuery(
    "update CreditCard c set c.stolenOn = :now where c.owner like 'J%'"
).setParameter("now", new Date());
```

← Hibernate 甚至会为此更新创建一个临时表

即使必须生成一些 SQL 语句或者需要将一些数据复制到临时表中，Hibernate 也知道如何执行这个更新；它会更新几个基表中的行（因为 CreditCard 被映射到了几个超类和子类表）。

JPQL UPDATE 语句仅能引用单个实体类，并且条件大批量操作可能只有一个根实体；例如，你不能编写单个语句来同时更新 Item 和 CreditCard 数据。WHERE 子句允许使用子查询，并且所有的联结都只在这些子查询中使用。

可以更新一个嵌入类型的值：例如，`update User u set u.homeAddress.street = ...`。你不能更新一个集合中可嵌入类型的值。这是不允许的：`update Item i set i.images.title = ...`。

Hibernate 特性

默认情况下，直接的 DML 操作不会影响被影响实体中的任何版本或时间戳（与 JPA 所标准化的一样）。但 Hibernate 扩展可以递增直接修改的实体实例的版本号：

```
int updatedEntities =
    em.createQuery("update versioned Item i set i.active = true")
        .executeUpdate();
```

现在每一个更新过的 Item 实体实例的版本在数据库中都会被直接递增，任何其他乐观并发控制的事务表明你修改了数据（如果版本或时间戳属性依赖 `custom org.hibernate.usertype.UserVersionType`，Hibernate 就不允许使用 `versioned` 关键字）。

使用 JPA 条件 API，你就必须自己递增版本：

```
CriteriaUpdate<Item> update =
    criteriaBuilder.createCriteriaUpdate(Item.class);

Root<Item> i = update.from(Item.class);

update.set(i.get(Item_.active), true);
update.set(
    i.get(Item_.version),
    criteriaBuilder.sum(i.get(Item_.version), 1)
);

int updatedEntities = em.createQuery(update).executeUpdate();
```

我们介绍的第二个批量操作是 DELETE：

```
em.createQuery("delete CreditCard c where c.owner like 'J%'")
    .executeUpdate();
CriteriaDelete<CreditCard> delete =
    criteriaBuilder.createCriteriaDelete(CreditCard.class);

Root<CreditCard> c = delete.from(CreditCard.class);

delete.where(
    criteriaBuilder.like(
        c.get(CreditCard_.owner),
        "J%"
    )
);

em.createQuery(delete).executeUpdate();
```

用于 UPDATE 语句和 CriteriaUpdate 的相同规则适用于 DELETE 和 CriteriaDelete：在 WHERE 子句中不允许使用任何联结、单一实体类、可选的别名或者子查询。

另一个特殊的 JPQL 批量操作可以直接在数据库中创建实体实例。

Hibernate 特性

创建新的实体实例

我们假设你的一些顾客的信用卡已经被盗。你要编写两个批量操作来标记他们被盗的日期(其实也就是你发现被盗的日期)，并且要将受威胁的信用卡数据从你的记录中移除。由于你是在为一家负责任的公司工作，因此必须将被盗的信用卡报告给当局和受影响的顾客。因此，在你删除这些记录之前，你要提取被盗的所有记录并且创建几百(或几千)条 StolenCreditCard 记录。仅仅为此，你就要编写一个新的映射实体类：

@Entity

```
public class StolenCreditCard {
```

@Id

```
public Long id;
```

← 应用程序指定的


```

public String owner;
public String cardNumber;
public String expMonth;
public String expYear;
public Long userId;
public String username;
public StolenCreditCard() {
}

public StolenCreditCard(Long id,
                          String owner, String cardNumber,
                          String expMonth, String expYear,
                          Long userId, String username) {
}
}

```

Hibernate 会将这个类映射到 `STOLENCREDITCARD` 表。接下来，需要一个在数据库中直接执行的语句，检索所有受威胁的信用卡，并且创建新的 `StolenCreditCard` 记录。这可以使用仅用于 Hibernate 的 `INSERT...SELECT` 语句：

```

int createdRecords =
    em.createQuery(
        "insert into" +
        " StolenCreditCard(id, owner, cardNumber, expMonth, expYear,
        ➤userId, username)" +
        " select c.id, c.owner, c.cardNumber, c.expMonth, c.expYear, u.id,
        ➤u.username" +
        " from CreditCard c join c.user u where c.owner like 'J%'"
    ).executeUpdate();

```

这个操作会做两件事情。首先，它会选取 `CreditCard` 记录的详情以及它们各自的所有者(一个 `User`)。其次，它会将结果直接插入到由 `StolenCreditCard` 类映射的表中。

注意以下事情：

- `INSERT ... SELECT` 的目标属性(在这个例子中是你列出的 `StolenCreditCard` 属性)必须被用于某个特定子类，而非一个(抽象)超类。由于 `StolenCreditCard` 不是继承性层次结构的一部分，所以这样做没什么问题。
- 由 `SELECT` 中投影所返回的类型必须匹配 `INSERT` 参数所要求的类型。
- 在该示例中，`StolenCreditCard` 的标识符属性位于插入属性的列表中，并且是通过选择来提供的；它与原始的 `CreditCard` 标识符值相同。或者，可以为 `StolenCreditCard` 映射一个标识符生成器；但这仅适用于直接在数据库内部操作的标识符生成器，比如序列或标识字段。
- 如果所生成的记录是一个版本化的类(具有一个版本或时间戳属性)，那么还会生成一个新的版本(零或当前时间戳)。或者，可以选择一个版本(或时间戳)值并且将该版本(或时间戳)属性添加到插入属性的列表。

在编写本书时，JPA 或 Hibernate 条件 API 还不支持 `INSERT ... SELECT` 语句。

JPQL 和条件大批量操作能满足许多你通常要借助普通 SQL 来应对的情况的需求。在某些情况下，你可能希望执行 SQL 大批量操作，而不是回退到 JDBC。

20.1.2 SQL 中的大批量语句

在上一节中，你看到了 JPQL UPDATE 和 DELETE 语句。这些语句的主要优势在于，它们适用于类和属性名称，并且 Hibernate 知道如何在生成 SQL 时处理继承性层次结构和版本控制。由于 Hibernate 会解析 JPQL，因此它也知道如何在查询之前有效地进行脏检查并且刷新持久化上下文，以及如何无效化二级缓存区域。

如果 JPQL 没有你所需的功能，那么可以执行原生的 SQL 大批量语句：

```
Query query = em.createNativeQuery(
    "update ITEM set ACTIVE = true where SELLER_ID = :sellerId"
).setParameter("sellerId", johndoe.getId()); 所有的二级缓存区域都被清除了
int updatedEntities = query.executeUpdate();
assertEquals(updatedEntities, 2);
```

更新行，而非更新实体实例

使用 JPA 原生大批量语句，就必须意识到一个重要问题：Hibernate 不会解析 SQL 语句来检测受影响的表。这意味着 Hibernate 不知道在查询执行之前是否需要刷新持久化上下文。在上一个示例中，Hibernate 不知道正在更新 ITEM 表中的行。Hibernate 必须在执行查询时进行脏检查并且刷新持久化上下文中的所有实体实例；它不能仅对持久化上下文中的 Item 实例进行脏检查和刷新。

如果启用二级缓存，就必须思考另一个问题(如果不启用，就无须担心它)：Hibernate 必须保持二级缓存同步以避免返回过期数据，因此它会在你执行原生 SQL UPDATE 或 DELETE 语句时无效化和清除所有二级缓存区域。这意味着你的二级缓存存在这个查询之后将会变空！

Hibernate 特性

可以将 Hibernate API 用于 SQL 查询来得到对脏检查、刷新和二级缓存无效化的更细粒度的控制：

```
org.hibernate.SQLQuery query =
    em.unwrap(org.hibernate.Session.class).createSQLQuery(
        "update ITEM set ACTIVE = true where SELLER_ID = :sellerId"
    );
query.setParameter("sellerId", johndoe.getId());
query.addSynchronizedEntityClass(Item.class);
int updatedEntities = query.executeUpdate();
assertEquals(updatedEntities, 2);
```

只会清除具有 Item 数据的二级缓存区域

更新行，而非更新实体实例

使用 addSynchronizedEntityClass() 方法，就可以让 Hibernate 知道 SQL 语句影响了哪些

表，而 Hibernate 将只清除相关的缓存区域。Hibernate 现在还知道，在查询之前，它只需要刷新持久化上下文中修改过的 Item 实体实例。

有时无法在大量数据操作中排除应用层。必须将数据加载到应用程序内存中，并且使用 EntityManager 来执行更新和删除，这就需要用到我们将要介绍的批处理。

20.1.3 批处理

如果必须在一个事务和工作单元中创建或更新几百或几千个实体实例，那么可能会耗尽内存。此外，还必须考虑用于完成事务的时间。大多数事务管理器都有一个数秒或数分钟范围的低事务超时设置。用于本书示例的 Bitronix 事务管理器具有 60 秒的默认事务超时设置。如果你的工作单元需要更长的时间来完成，那么应该首先为特定事务重写此超时设置：

```
tx.setTransactionTimeout(300); ← 5 分钟
```

这就是 UserTransaction API。只有在这个线程上开启的未来的事务才使用该新的超时设置。必须在 begin() 事务之前设置此超时。

接下来，我们将几千个 Item 实例批量插入到数据库中。

批量插入实体实例

你传递到 EntityManager#persist() 的每一个瞬时实体实例都会被添加到持久化上下文缓存中，就像 10.2.8 节中所阐释的一样。为了避免内存耗尽，你要在特定数量的插入之后对该持久化上下文执行 flush() 和 clear() 操作，这实际上就是批量处理插入。见代码清单 20.2。

代码清单 20.2 插入大量实体实例

```
tx.begin();
EntityManager em = JPA.createEntityManager();

for (int i = 0; i < ONE_HUNDRED_THOUSAND; i++) { ← ❶ 创建实例
    Item item = new Item(
        // ...
    );
    em.persist(item);

    if (i % 100 == 0) { ← ❷ 多次执行 INSERT
        em.flush();
        em.clear();
    }
}

tx.commit();
em.close();
```

❶ 创建并且持久化 100 000 个 Item 实例。

- ② 在 100 个操作之后，刷新和清除持久化上下文。这样就会为 100 个 Item 实例执行 SQL INSERT 语句，并且由于它们现在处于分离状态且不再被引用，因此垃圾回收可以收回该内存。

应该将持久化单元中的 `hibernate.jdbc.batch_size` 属性设置为与批量大小相同的数值，这里是 100。使用此设置，Hibernate 就会在 JDBC 级别使用 `PreparedStatement#addBatch()` 批量处理 INSERT 语句。

批量处理交错的 SQL 语句

以交错方式持久化几种不同实体实例的批量处理过程，比如一个 Item，然后一个 User，然后另一个 Item，另一个 User，以此类推，实际上这种并非是 JDBC 级别的批量处理。在刷新时，Hibernate 会生成一个 `insert into ITEM` SQL 语句，然后生成一个 `insert into USERS` 语句，然后生成另一个 `insert into ITEM` 语句，以此类推。如果每一个语句都与上一个语句不同的话，那么 Hibernate 就不能一次性执行较大的批量处理。如果在持久化单元配置中启用属性 `hibernate.order_inserts`，那么 Hibernate 就会在尝试构建一个批次的语句之前对操作排序。然后 Hibernate 会为 ITEM 表执行所有的 INSERT 语句，并且为 USERS 表执行所有的 INSERT 语句。之后，Hibernate 可以在 JDBC 级别批量处理语句。

如果为 Item 实体启用共享的二级缓存，那么就应该避开用于批量(插入)过程的缓存；参阅 20.2.5 节。

使用大量插入的一个严重问题在于，对标识符生成器的竞争：`EntityManager#persist()` 的每个调用都必须获得一个新的标识符值。通常，该生成器是一个数据库序列，为每一个持久化的实体实例调用一次。为了得到高效的批处理，必须减少与数据库交互的次数。

Hibernate 特性

在 4.2.5 节中，我们推荐了专用于 Hibernate 的 `enhanced-sequence` 生成器，尤其是因为它支持适用于批量操作的某些优化。首先，在 `package-info.java` 元数据中定义该生成器：

```
@org.hibernate.annotations.GenericGenerator(
    name = "ID_GENERATOR_POOLED",
    strategy = "enhanced-sequence",
    parameters = {
        @org.hibernate.annotations.Parameter(
            name = "sequence_name",
            value = "JPWH_SEQUENCE"
        ),
        @org.hibernate.annotations.Parameter(
            name = "increment_size",
            value = "100"
        ),
        @org.hibernate.annotations.Parameter(
            name = "optimizer",
            value = "pooled-lo"
        )
    }
)
```

现在在映射的实体类中用 `@GeneratedValue` 使用该生成器。

将 `increment_size` 设置为 100，序列会生成“接下来的”值 100、200、300、400 等。Hibernate 中的 `pooled-lo` 优化器会在你每次调用 `persist()` 时生成中间值，而无须对数据库进行另一次交互。因此，如果从该序列获得的下一个值是 100，那么 Hibernate 就会在应用层中生成标识符值 101、102、103 等。一旦耗尽了该优化器的 100 个标识符值这个池，数据库就会获得下一个序列值，并且处理过程会重复进行。这意味着每 100 个插入的批次，你只需要交互一次就能从数据库中得到一个标识符值。其他的标识符生成优化器也是可用的，但 `pooled-lo` 优化器几乎覆盖了所有的用例，并且最易于理解和配置。

要注意，如果一个应用程序使用了相同的序列却没有应用与 Hibernate 优化器相同的算法，那么 100 这个增量大小就会在数值标识符之间留下间隙。不必对此太过关注；相较于能够为 3 亿年中的每一毫秒生成一个新的标识符值，在 3 百万年中你可能就会耗尽其数字空间。

可以使用相同的批量处理技术来更新大量的实体实例。

Hibernate 特性

批量更新实体实例

假设必须处理许多 Item 实体实例并且需要做出的修改并不像设置一个标记那么简单(你之前使用单个 UPDATE JPQL 语句完成了该设置)。我们再假设出于某些原因，你不能创建一个数据库存储过程(可能是由于你的应用程序必须工作在不支持存储过程的数据库管理系统上)。唯一的一个选择就是在 Java 中编写程序并且将大量的数据获取到内存中以便通过该程序来运行它。

这就要求批量处理并且使用一个数据库游标滚动浏览查询结果，这是一个 Hibernate 专用的查询功能。请回顾一下 14.3.3 节中关于使用游标进行滚动的阐释，并且确保你的 DBMS 和 JDBC 驱动正常支持数据库游标。代码清单 20.3 所示的代码会一次性加载 100 个 Item 实体实例用于处理。

代码清单20.3 更新大量的实体实例

```
tx.begin();
EntityManager em = JPA.createEntityManager();

org.hibernate.ScrollableResults itemCursor = ← ① 打开游标
    em.unwrap(org.hibernate.Session.class)
        .createQuery("select i from Item i")
        .scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

int count = 0;
while (itemCursor.next()) { ← ② 移动游标
    Item item = (Item) itemCursor.get(0); ← ③ 检索实例
    modifyItem(item);

    if (++count % 100 == 0) { ← ④ 刷新持久化上下文
        em.flush();
        em.clear();
    }
}
```

```

    }
}

```

```

itemCursor.close();
tx.commit();
em.close();

```

- ❶ 使用一个 JPQL 查询从数据库加载所有的 Item 实例。打开一个在线数据库游标，而不是将查询结果完全检索到应用程序内存中。
- ❷ 使用 ScrollableResults API 控制该游标并且沿着结果移动它。对 next() 的每个调用都会将该游标前进到下一个记录。
- ❸ get(int i) 调用会将单个实体实例获取到内存中：游标当前指向的记录。
- ❹ 为了避免内存耗尽，要在将后 100 条记录加载到其中之前刷新并且清除持久化上下文。

为了得到最佳性能，应该将持久化上下文单元配置中的属性 `hibernate.jdbc.batch_size` 的大小设置为与你的程序批处理相同的值：100。Hibernate 会在 JDBC 级别批量处理刷新时执行的所有 UPDATE 语句。默认情况下，如果已经为一个实体类启用了版本管理，那么 Hibernate 就不会在 JDBC 级别进行批处理——一些 JDBC 驱动在返回批量 UPDATE 语句的正确更新行数方面存在问题(Oracle 就有这个问题)。如果确定你的 JDBC 驱动正常支持这个功能，并且你的 Item 实体类有一个 `@Version` 注解，就可以通过将属性 `hibernate.jdbc.batch_versioned_data` 设置为 `true` 来启用 JDBC 批处理。如果为 Item 实体启用共享二级缓存，那么应该避开用于批处理(更新)程序的缓存；请参阅 20.2.5 节。

在持久化上下文中避免内存消耗的另一个选项(实际上是通过禁用它)是 `org.hibernate.StatelessSession` 接口。

20.1.4 Hibernate StatelessSession 接口

持久化上下文是 Hibernate 引擎的一个必要功能。不使用持久化上下文的话，你就无法处理实体状态并且让 Hibernate 自动检测你的变更。其他许多事情也就无法进行了。

不过，如果更愿意通过执行语句来使用你的数据库的话，那么 Hibernate 提供了一个可替代的接口。这个面向语句的接口就是 `org.hibernate.StatelessSession`，它看起来很像普通的 JDBC，运行起来也很像，只是你能得到映射持久化类的好处以及 Hibernate 的数据库移植性。这个接口中最吸引人的方法是 `insert()`、`update()` 和 `delete()`，它们都映射到了等效的立即执行的 JDBC/SQL 操作。

我们使用这个接口来编写与之前示例中相同的“更新所有商品实体数据”的程序。见代码清单 20.4。

代码清单 20.4 使用一个 StatelessSession 更新数据

```

tx.begin();

org.hibernate.SessionFactory sf = ← ❶ 开启 StatelessSession
    JPA.getEntityManagerFactory().unwrap(org.hibernate.SessionFactory.class);

```



```

org.hibernate.StatelessSession statelessSession = sf.openStatelessSession();
org.hibernate.ScrollableResults itemCursor = ← ② 加载 Item 实例
    statelessSession
        .createQuery("select i from Item i")
        .scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);
while (itemCursor.next()) { ← ③ 检索实例
    Item item = (Item) itemCursor.get(0);

    modifyItem(item);

    statelessSession.update(item); ← ④ 执行 UPDATE
}

itemCursor.close();
tx.commit();
statelessSession.close();

```

- ① 在 Hibernate SessionFactory 上开启一个 StatelessSession, 可以从一个 EntityManagerFactory 中解包它。
- ② 使用一个 JPQL 查询从数据库加载所有的 Item 实例。要打开一个在线数据库游标, 而不是将查询结果完全检索到应用程序内存中。
- ③ 使用游标滚动结果, 并且检索一个 Item 实体实例。这个实例处于分离状态; 这里没有持久化上下文!
- ④ 由于 Hibernate 不会在没有持久化上下文的情况下自动检测变更, 因此你必须手动执行 SQL UPDATE 语句。

禁用持久化上下文以及使用 StatelessSession 接口有其他一些严重后果和概念上的局限 (至少, 如果将它与常规的 EntityManager 和 org.hibernate.Session 做对比的话, 就是如此):

- StatelessSession 没有持久化上下文缓存并且不会与任何其他二级缓存或查询缓存交互。在事务提交时, 没有自动化的脏检查或 SQL 执行。你所做的一切都会引发即时的 SQL 操作。
- 实体实例的修改以及你调用的操作都不会被级联到任何关联实例。Hibernate 会忽略映射中的所有级联规则。你是在处理单个实体类的实例。
- 你没有受保障的对象标识作用域。在相同 StatelessSession 中执行相同查询两次会产生两个不同的内存中分离的实例。如果在持久化类中不谨慎实现 equals() 和 hashCode() 方法, 这就会导致数据混淆的影响。
- Hibernate 会忽略对映射为实体关联 (一对多、多对多) 的集合所做的任何修改。只会处理基本集合或可嵌入类型。因此, 不应该使用多对多实体关联——而是仅使用多对一或一对一关联——并且处理仅通过那一端的关系。你要编写一个查询来获取数据, 否则就只能通过遍历映射集合来检索那些数据。
- Hibernate 不会为使用 StatelessSession 执行的操作而调用 JPA 事件侦听器和事件回调方法。StatelessSession 会避开任何启用的 org.hibernate.Interceptor, 无法通过 Hibernate 核心事件系统来拦截它。

用于 StatelessSession 的好用例很少; 如果使用常规 EntityManager 手动批处理变得复

杂繁琐，那么你可能会选择它。

在下一节中，我们将介绍 Hibernate 共享缓存系统。在应用层缓存数据是一个互补的优化，可以在任何复杂的多用户应用程序中使用它。

20.2 缓存数据

在本节中，我们将介绍如何启用、调整以及管理 Hibernate 中的共享数据缓存。共享数据缓存并非持久化上下文缓存，Hibernate 不会在应用程序线程之间共享持久化上下文缓存。由于 10.1.2 节中阐释的原因，这不是可选的。我们将持久化上下文称为一级缓存。共享的数据缓存——二级缓存——是可选的，尽管 JPA 标准化了一些配置设置以及用于共享缓存的映射元数据，但每个供应商都有用来优化的不同的解决方案。我们首先介绍一些背景信息并且探究 Hibernate 共享缓存的架构。

Hibernate 特性

20.2.1 Hibernate 共享的缓存架构

缓存会在应用程序服务器的内存或硬盘上保留接近于应用程序的当前数据库状态的表示。缓存是数据的本地副本并且位于应用程序和数据库之间。简单来说，对于 Hibernate，缓存看起来就像是一个键/值对的映射。Hibernate 可以通过提供一个键和一个值来将数据存储在缓存中，并且它可以使用键在缓存中查找值。

Hibernate 有几种共享缓存的类型可用。在以下情况发生时，可以使用一个缓存来避免数据库访问：

- 应用程序会根据标识符(主键)来执行一个实体实例查找；这可能会访问一次实体数据缓存。按需初始化一个实体代理是相同的操作，并且从内部来说，可能会访问实体数据而非数据库。缓存键是实体实例的标识符值，而缓存值是实体实例的数据(其属性值)。实际的数据是以特殊的分解格式来存储的，而 Hibernate 会在它读取实体数据缓存时再次整合一个实体实例。
- 持久化引擎会延迟初始化一个集合；集合缓存可以持有该集合的元素。缓存键是集合的角色：例如，“Item[1234]#bids”可以是标识符为 1234 的 Item 实例的 bids 集合。这个例子中的缓存值是一组 Bid 标识符值，即集合的元素(注意，这个集合缓存没有持有 Bid 实体数据，仅持有了数据的标识符值！)
- 应用程序会根据唯一键属性执行实体实例查找。这是用于具有唯一属性的实体类的特殊自然标识符缓存：比如 User#username。缓存键是一个唯一属性，比如 username，而缓存的值是 User 的实体实例标识符。
- 要执行一个 JPQL、条件或 SQL 查询，并且实际的 SQL 查询的结果已经被存储在查询结果缓存中。该缓存键是所呈现的包含其所有参数值的 SQL 语句，而缓存值是 SQL 结果集的一些表示，其中可能包含实体标识符值。

理解实体数据缓存是持有实际实体数据值的唯一一种缓存类型至关重要。其他三种缓

存类型仅持有实体标识符信息。因此，启用自然标识符缓存没什么意义，比如，在不启用实体数据缓存的情况下。当找到匹配的时候，在自然标识符缓存中进行查找总是会涉及在实体数据缓存中进行查找。接下来我们将使用一些代码示例进一步分析此行为。

正如我们之前提示过的，Hibernate 具有二级缓存架构。

为不可变数据启用引用存储

Hibernate 会以分解的格式将二级实体缓存中的数据作为一个副本来持有，并且在从缓存中读取时重新整合它。复制数据是一个开销很大的操作；因此，作为一种优化，Hibernate 允许指定不可变的数据可以被原样存储，而不是复制到二级缓存中。这可用于引用数据。我们假设你有一个具有属性 `zipcode` 和 `name` 的 `City` 实体类，使用了 `@Immutable` 注解。如果在持久化单元中启用配置属性 `hibernate.cache.use_reference_entries`，那么 Hibernate 将会尝试（并且在某些特殊情况下无法这么做）在二级数据缓存中直接存储 `City` 的一个引用。这里要注意的是，如果在应用程序中偶然修改了 `City` 的一个实例，那么该变更实际上将写到（本地）缓存区域的所有并发用户，因为他们都会得到相同的引用。

二级缓存

可以在图 20-1 中看到 Hibernate 缓存系统的各个元素。一级缓存是持久化上下文缓存，我们在 10.1.2 节中探讨过。Hibernate 没有在线程之间共享此缓存；在此缓存中每个应用程序线程都有其自己的数据副本。因此，在访问这个缓存时事务隔离和并发方面没有任何问题。

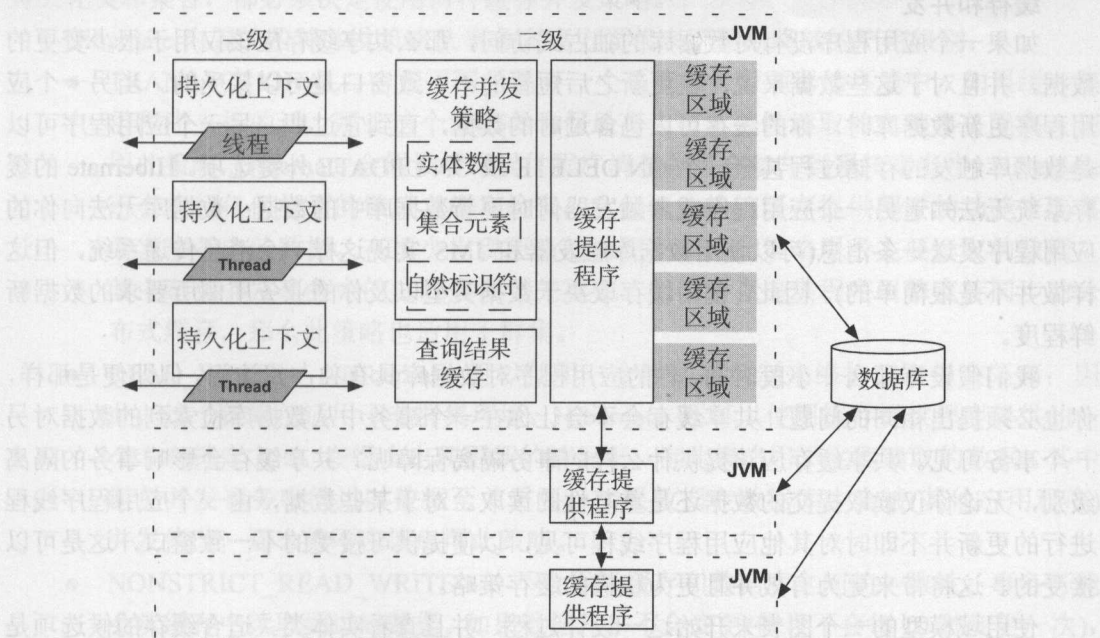


图 20-1 Hibernate 的二级缓存架构

Hibernate 中的二级缓存系统在 JVM 中可能是进程作用域的或者可能是一个可以工作在 JVM 群集中的缓存系统。多应用程序线程可能会并发访问共享二级缓存。缓存并发策略会定义用于实体数据、集合元素以及自然标识符缓存的事务隔离详情。无论何时在这些缓

存中存储或加载一个条目, Hibernate 都会使用配置的策略调整访问。为实体类及其集合选取合适的缓存并发策略会充满挑战, 并且我们稍后将用几个示例指导你完成该处理过程。

查询结果缓存也有其自己的、内部的策略用于处理并发访问并且保持缓存的结果为最新, 以及与数据库协调一致。我们将介绍查询缓存如何工作以及对于哪些查询来说启用结果缓存是合理的。

缓存提供程序将物理缓存实现为可插拔系统。目前, Hibernate 会强制你为整个持久化单元选择单一缓存提供程序。该缓存提供程序要负责处理物理缓存区域——应用层上持有数据的位置(内存中、索引文件中, 甚至在群集中复制)。该缓存提供程序会控制过期策略, 比如根据超时从一个区域移除数据时, 或者当缓存满了时仅保持最近使用过的数据。缓存提供程序实现或许能够与 JVM 群集中的其他实例通信, 以便同步每个实例存储位置中的数据。Hibernate 本身不会处理任何缓存群集; 这是完全委托给缓存提供程序引擎的。

在这一节中, 你要在单个 JVM 上使用 Ehcache 提供程序设置缓存, 该提供程序是一个简单但非常强大的缓存引擎(最初是作为易用型 Hibernate 缓存特别为 Hibernate 开发的)。我们仅介绍 Ehcache 的一些基础设置; 可以参考其手册以便获得更多信息。

许多开发人员对于 Hibernate 缓存系统常常提出的第一个问题是, “当数据在数据库中被修改时, 缓存会知道吗?” 在你开始着手处理缓存配置和使用之前, 我们来尝试回答这个问题。

缓存和并发

如果一个应用程序没有对数据库的独占式访问, 那么共享缓存应该仅用于很少变更的数据, 并且对于这些数据来说, 在更新之后短暂的不一致窗口是可以接受的。当另一个应用程序更新数据库时, 你的缓存可以包含过时的数据, 直到它过期。另一个应用程序可以是数据库触发的存储过程甚至一个 ON DELETE 或 ON UPDATE 外键选项。Hibernate 的缓存系统无法知道另一个应用程序或者触发器何时更新数据库中的数据; 数据库无法向你的应用程序发送一条消息(可以使用数据库触发器和 JMS 实现这样一个消息传递系统, 但这样做并不是很简单的)。因此, 使用缓存取决于数据类型以及你的业务用例所要求的数据新鲜程度。

我们假设有那么一小段时间, 你的应用程序对数据库具有独占式访问。但即便是那样, 你也必须提出相同的问题, 共享缓存会不会让你在一个事务中从数据库检索到的数据对另一个事务可见。共享缓存应该提供什么样的事务隔离保障呢? 共享缓存会影响事务的隔离级别, 无论你仅读取提交的数据还是重复性的读取。对于某些数据, 由一个应用程序线程进行的更新并不即时对其他应用程序线程可见, 以便提供可接受的不一致窗口, 这是可以接受的。这将带来更为有效并且更为积极的缓存策略。

使用域模型的一个图表来开始这一设计过程, 并且查看实体类。适合缓存的候选项是表示以下内容的类:

- 很少变更的数据
- 非关键性数据(例如, 内容管理数据)
- 应用程序的本地数据并且不会被其他应用程序所修改

不合适的候选项包括:

- 经常更新的数据
- 财务数据，其中决策必须要基于最新的更新
- 被其他应用程序共享和/或写入的数据

没有哪些规则是我们通常会使用的。许多应用程序都有具有以下属性的若干个类：

- 全部适合放入内存中的少量实例(数千个，而非数百万个)
- 被另一个或多个类的许多实例所引用的每一个实例
- 很少(或永远不会)更新的实例

有时候我们将此类数据称为引用数据。引用数据的示例包括邮编、位置、静态文本消息等。引用数据是适合共享缓存的绝佳候选项，任何深度使用引用数据的应用程序都能从缓存该数据中获得极大的好处。当缓存超时过期时，要允许数据被刷新，并且在更新之后短暂的不一致窗口是可以接受的。实际上，有些引用数据(比如国家代码)可以具有很长的不一致窗口或者可以在数据是只读时被永久缓存。

你必须练习对希望启用缓存的每一个类和集合的仔细评估。你必须决定使用何种并发策略。

选择一种缓存并发策略

缓存并发策略是一个中介者：它负责在缓存中存储数据项并且从缓存中检索它们。这个重要角色定义了用于特定项的事务隔离语义。如果希望启用共享缓存，那么对于每一个持久化类和集合，都必须决定使用何种缓存并发策略。

Hibernate 内置的四种并发策略从事务隔离方面代表了逐渐降低的严密级别：

- **TRANSACTIONAL**——仅可用于具有一个系统事务管理器的环境中，如果缓存提供程序支持的话，那么这个策略会确保最高为可重复性读取的完整事务隔离。使用此策略，Hibernate 会假设缓存提供程序清楚并且会参与到系统事务中。Hibernate 不会执行任何类型的锁或者版本检查；它仅仅依赖该缓存提供程序的功能来隔离并发事务中的数据。在很少出现更新的情况下，将此策略用于主要读取的数据，其中避免并发事务中的过时数据至关重要。如果缓存提供程序引擎支持同步的分布式缓存，那么此策略也适用于群集。
- **READ_WRITE**——Hibernate 可以使用一个时间戳机制来维护读取提交隔离；因此，这个策略仅适用于非群集环境。Hibernate 还可以使用缓存提供程序提供的一个专有锁定 API。在很少出现更新的情况下，将此策略用于主要读取的数据，其中避免并发事务中的过时数据至关重要。如果数据在数据库中会被(其他应用程序)并发修改，那么就不应该启用此策略。
- **NONSTRICT_READ_WRITE**——不会保障缓存和数据库之间的一致性。事务可能会从缓存中读取到过时数据。如果数据几乎不会变更(假设不会每 10 秒变更一次)，并且不一致窗口并非关键问题的话，则可以使用此策略。要使用缓存提供程序的过期策略来配置不一致窗口的时间周期。此策略可用于群集中，甚至可用于异步分布式缓存。如果有其他应用程序在相同数据库中修改数据，那么它可能就是合适的。

- **READ_ONLY**——适用于绝不会变更的数据。如果触发一次更新，则会得到一个异常。仅将它用于引用数据。

使用逐渐降低的严密性会带来性能和可扩展性的提升。使用 **NONSTRICT_READ_WRITE** 的群集异步缓存可以处理的事务量远比使用 **TRANSACTIONAL** 的同步群集要多。在用于生产环境之前，必须谨慎估算使用完整事务隔离的群集缓存的性能。在许多情况下，如果不能出现过时数据，则最好不要为特定类启用共享缓存！

应该禁用共享缓存来对应用程序进行基准测试。一次仅为一个合适的候选类启用共享缓存，同时持续测试系统的可扩展性并且评估并发策略。必须使用可用的自动化测试来判断对缓存设置进行变更所带来的影响。为了应用程序的性能和可扩展性关注点，我们建议你在启用共享缓存之前首先编写这些测试。

在你了解了这一原理的全部信息之后，是时候看看在实践中缓存是如何工作的了。首先，要配置共享缓存。

20.2.2 配置共享缓存

在 `persistence.xml` 配置文件中配置共享缓存。见代码清单 20.5。

代码清单20.5 persistence.xml中的共享缓存配置
路径: /model/src/main/resources/META-INF/persistence.xml

```
<persistence-unit name="CachePU">
    ...
    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
    <properties>
        <property name="hibernate.cache.use_second_level_cache"
            value="true"/>
        <property name="hibernate.cache.use_query_cache"
            value="true"/>
        <property name="hibernate.cache.region.factory_class"
            value="org.hibernate.cache.ehcache.
                SingletonEhCacheRegionFactory"/>
        <property name="net.sf.ehcache.configurationResourceName"
            value="/cache/ehcache.xml"/>
        <property name="hibernate.cache.use_structured_entries"
            value="false"/>
        <property name="hibernate.generate_statistics"
            value="true"/>
    </properties>
</persistence-unit>
```

- ❶ 共享缓存模式会控制这个持久化单元的实体类变成可缓存的方式。通常更愿意选择性地仅为某些实体类启用缓存。选项有：**DISABLE_SELECTIVE**、**ALL** 和 **NONE**。
- ❷ **Hibernate** 的二级缓存系统必须被显式启用；默认情况下不会启用它。可以单独启用查询结果缓存；默认情况下它也是被禁用的。

- ③ 为二级缓存系统选取一个提供程序。对于 Ehcache, 要将 `org.hibernate:hibernate-ehcache` Maven 构件依赖添加到类路径。然后, 使用此区域工厂设置选择 Hibernate 如何使用 Ehcache; 这里你告知了 Hibernate 将单个 Ehcache 实例在内部作为二级缓存提供程序来管理。
- ④ 当该提供程序启动时, Hibernate 会将这个属性传递给 Ehcache, 设置 Ehcache 配置文件的位置。用于缓存区域的所有物理缓存设置都在这个文件中。
- ⑤ 当数据被存储以及从二级缓存中加载时, 这会控制 Hibernate 分解和装配实体状态的方式。结构化的缓存条目格式效率较低, 但在群集环境中是必要的。对于像这个 JVM 上的单例 Ehcache 这样的非群集二级缓存来说, 可以禁用此设置并且使用一个更为有效的格式。
- ⑥ 当试用二级缓存时, 你通常会希望看到屏幕背后正在发生什么。Hibernate 有一个统计收集器和一个 API 来访问这些统计数据。出于性能原因, 默认情况下它是被禁用的(并且应该在生产环境中禁用它)。

二级缓存系统现在就准备好了, 并且 Hibernate 会在你为这个持久化单元构建一个 `EntityManagerFactory` 时启动 Ehcache。不过, Hibernate 默认不会缓存任何内容; 必须为实体类及其集合选择性地启用缓存。

20.2.3 启用实体和集合缓存

现在我们来查看 `CaveatEmptor` 域模型的实体类和集合, 并且使用正确的并发策略启用缓存。同时, 要在 Ehcache 配置文件中配置必要的物理缓存区域。

首先处理 `User` 实体: 这个数据很少变更, 不过当然, 用户可能会经常修改它们的用户名或地址。这从财务上来说并非关键数据; 没人会基于一个用户的名称或地址来决定购买行为。当用户修改名称或地址信息时, 短暂的不一致窗口是可以接受的。我们假设对于最大一分钟这个时长来说, 这是没有问题的, 老的信息仍旧在某些事务中可见。这意味着可以使用 `NONSTRICT_READ_WRITE` 策略启用缓存:

路径: `/model/src/main/java/org/jpwh/model/cache/User.java`

```
@Entity
@Table(name = "USERS")
@Cacheable
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations
        .CacheConcurrencyStrategy.NONSTRICT_READ_WRITE,
    region = "org.jpwh.model.cache.User"
)
@org.hibernate.annotations.NaturalIdCache
public class User {

    @NotNull
    @org.hibernate.annotations.NaturalId(mutable = true)
    @Column(nullable = false)
    protected String username;
```

← 默认名称

由于@NaturalId 而忽略模式生成

← 让它变成 UNIQUE

← 用于模式生成

Hibernate 特性

@Cacheable 注解会为这个实体类启用共享缓存，但 Hibernate 注解对于选取并发策略是必要的。Hibernate 会在名称为 your.package.name.User 的一个缓存区域的二级缓存中存储和加载 User 实体数据。可以使用 @Cache 注解的 region 属性重写该名称(或者，可以在持久化单元中使用 hibernate.cache.region_prefix 属性设置一个全局的区域名称前缀)。

你还要使用 @org.hibernate.annotations.NaturalIdCache 为 User 实体启用自然标识符缓存。该自然标识符属性是使用 @org.hibernate.annotations.NaturalId 来标记的，并且必须告知 Hibernate 该属性是否可变。这样就让你能够根据 username 查找 User 实例，而无须访问数据库。

接下来，要在 Ehcache 中为实体数据和自然标识符缓存配置缓存区域：

路径：/model/src/main/resources/cache/ehcache.xml

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">

    <cache name="org.jpwh.model.cache.User"
        maxElementsInMemory="500"
        eternal="false"
        timeToIdleSeconds="30"
        timeToLiveSeconds="60"/>

    <cache name="org.jpwh.model.cache.User##NaturalId"
        maxElementsInMemory="500"
        eternal="false"
        timeToIdleSeconds="30"
        timeToLiveSeconds="60"/>

</ehcache>
```

可以在这两个缓存的每一个中存储最大 500 个条目，并且 Ehcache 不会永久保留它们。如果一个元素在 30 秒内没有被访问，那么 Ehcache 将移除它，并且甚至会在 1 分钟之后移除活动访问的条目。这会确保从缓存读取的不一致窗口绝不会超过 1 分钟。换句话说，缓存区域将保留最多 500 个最近使用过的用户账户，它们都是在最近 1 分钟内使用过的，并且会自动收缩。

我们继续处理 Item 实体类。这个数据经常变更，尽管你的读取仍旧远大于写入。如果一个商品的名称或描述被修改，那么并发事务就应该即时看到此更新。用户会基于商品的描述来进行财务决策是否要购买一个商品。因此，READ_WRITE 是一个合适的策略：

路径：/model/src/main/java/org.jpwh/model/cache/Item.java

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
```

```

}
public class Item {
    // ...
}

```

Hibernate 会在 Item 变更发生时调整读取和写入，以确保你总是可以从共享缓存读取提交的数据。如果另一个应用程序正在数据库中直接修改 Item 数据，那么所有的预期都会落空！要在 Ehcache 中配置缓存区域以便在一小时后将最近使用过的 Item 数据过期，以避免缓存区塞满过时的数据。

路径: /model/src/main/resources/cache/ehcache.xml

```

<cache name="org.jpwh.model.cache.Item"
    maxElementsInMemory="5000"
    eternal="false"
    timeToIdleSeconds="600"
    timeToLiveSeconds="3600"/>

```

思考一下 Item 实体类的 bids 集合: Item#bids 集合中的某个 Bid 是不可变的，但集合本身是可变的，而且并发工作单元需要即时看到集合元素的任何添加或移除:

路径: /model/src/main/java/org/jpwh/model/cache/Item.java

```

public class Item {
    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.Cache(
        usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
    )
    protected Set<Bid> bids = new HashSet<>();
    // ...
}

```

你要使用与拥有该集合的实体类的相同设置来配置缓存区域，因为每一个 Item 都有一个 bids 集合:

路径: /model/src/main/resources/cache/ehcache.xml

```

<cache name="org.jpwh.model.cache.Item.bids"
    maxElementsInMemory="5000"
    eternal="false"
    timeToIdleSeconds="600"
    timeToLiveSeconds="3600"/>

```

记住集合缓存不会包含实际的 Bid 数据至关重要。集合缓存仅会保留一组 Bid 标识符值。因此，必须也为 Bid 实体启用缓存。否则，Hibernate 就可能在开始遍历 Item#bids 时访问该缓存，不过，由于缓存未命中，Hibernate 会从数据库中单独加载每一个 Bid。这就是启用缓存会造成数据库服务器上产生更多负荷的情况！

我们已经说过，Bid 是不可变的，因此可以将这个实体数据缓存为 READ_ONLY:

路径: /model/src/main/java/org/jpwh/model/cache/Bid.java

```
@Entity
@org.hibernate.annotations.Immutable
@Cacheable
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY
)
public class Bid {
    // ...
}
```

尽管 Bid 是不可变的, 你也应该为缓存区域配置一个过期策略, 以避免旧的出价数据永久阻塞缓存:

路径: /model/src/main/resources/cache/ehcache.xml

```
<cache name="org.jpwh.model.cache.Bid"
    maxElementsInMemory="100000"
    eternal="false"
    timeToIdleSeconds="600"
    timeToLiveSeconds="3600"/>
```

现在就准备好测试缓存并且探究 Hibernate 的缓存行为了。

20.2.4 测试共享缓存

Hibernate 的透明缓存行为会难以分析。用于加载和存储数据的 API 仍然是 EntityManager, Hibernate 也会自动在缓存中写入和读取数据。当然, 可以通过记录 Hibernate 的 SQL 语句来看到实际的数据库访问, 但你应该让自己熟悉 org.hibernate.stat.Statistics API 以获得关于工作单元的更多信息并且获知屏幕背后正在发生什么。我们通过运行一些示例来看看这是如何工作的。

之前在 20.2.2 节中, 在持久化单元配置中启用了统计收集器。要在 org.hibernate.SessionFactory 上访问持久化单元的统计数据:

路径: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
Statistics stats =
    JPA.getEntityManagerFactory()
        .unwrap(SessionFactory.class)
        .getStatistics();

SecondLevelCacheStatistics itemCacheStats =
    stats.getSecondLevelCacheStatistics(Item.class.getName());
assertEquals(itemCacheStats.getElementCountInMemory(), 3);
assertEquals(itemCacheStats.getHitCount(), 0);
```

这里, 你还会得到 Item 实体的数据缓存区域的统计数据, 并且可以看到, 缓存中已经有几个实体存在了。这是一个热缓存; Hibernate 会在应用程序保存 Item 实体实例时在缓

存中存储数据。不过，实体还没有从缓存中被读取，并且访问次数为零。

当现在根据标识符查找一个 `Item` 实例时，Hibernate 会试图从缓存中读取数据并且避免执行一个 SQL SELECT 语句：

路径：/examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(itemCacheStats.getHitCount(), 1);
```

缓存中还有一些 `User` 实体数据，因此初始化 `Item#seller` 代理也会访问缓存：

路径：/examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
SecondLevelCacheStatistics userCacheStats =
    stats.getSecondLevelCacheStatistics(User.class.getName());
assertEquals(userCacheStats.getElementCountInMemory(), 3);
assertEquals(userCacheStats.getHitCount(), 0);

User seller = item.getSeller();
assertEquals(seller.getUsername(), "johndoe"); ← 初始化代理

assertEquals(userCacheStats.getHitCount(), 1);
```

当遍历 `Item#bids` 集合时，Hibernate 会使用该缓存：

路径：/examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
SecondLevelCacheStatistics bidsCacheStats =
    stats.getSecondLevelCacheStatistics(Item.class.getName() + ".bids");
assertEquals(bidsCacheStats.getElementCountInMemory(), 3);
assertEquals(bidsCacheStats.getHitCount(), 0);

SecondLevelCacheStatistics bidCacheStats =
    stats.getSecondLevelCacheStatistics(Bid.class.getName());
assertEquals(bidCacheStats.getElementCountInMemory(), 5);
assertEquals(bidCacheStats.getHitCount(), 0);

Set<Bid> bids = item.getBids();
assertEquals(bids.size(), 3);

assertEquals(bidsCacheStats.getHitCount(), 1);
assertEquals(bidCacheStats.getHitCount(), 3);
```

① 计算 `Item#bids` 集合的数量

② 计算 `Bids` 的数量

③ 读取缓存

④ 缓存结果

- ① 该统计数据会告诉你缓存中有 3 个 `Item#bids` 集合(每个 `Item` 一个)。目前还没有发生成功的缓存查找。
- ② `Bid` 的实体缓存具有 5 条记录，并且还没有访问过它。
- ③ 初始化该集合会从这两个缓存中读取数据。
- ④ 缓存发现了一个集合以及用于其 3 个 `Bid` 元素的数据。

`User` 的特殊自然标识符缓存并不完全是透明的。需要调用 `org.hibernate.Session` 上的一个方法来根据自然标识符执行查找：

路径: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
NaturalIdCacheStatistics userIdStats = ← ① 计算 User 数量
    stats.getNaturalIdCacheStatistics(User.class.getName() + "##NaturalId");
assertEquals(userIdStats.getElementCountInMemory(), 1);

User user = (User) session.byNaturalId(User.class) ← ② 自然标识符查找
    .using("username", "johndoe")
    .load();

assertNotNull(user); ← ③ 自然标识符访问

assertEquals(userIdStats.getHitCount(), 1); ←

SecondLevelCacheStatistics userStats = ← ④ 实体访问
    stats.getSecondLevelCacheStatistics(User.class.getName());
assertEquals(userStats.getHitCount(), 1);
```

- ① User 的自然标识符缓存区域有一个元素。
- ② org.hibernate.Session API 会执行自然标识符查找；这是用于访问自然标识符缓存的唯一 API。
- ③ 对于自然标识符查找，你有一次缓存访问。缓存返回了标识符值“johndoe”。
- ④ 对于该 User 的实体数据，你也有一次缓存访问。

统计 API 提供的信息远比我们在这些简单示例中所显示的要多；我们支持你进一步探究这个 API。Hibernate 会收集与其所有操作有关的信息，而且这些统计数据可用于查找热点，比如耗时最长的查询以及访问最多的实体和集合。

使用 JMX 访问统计数据

可以在运行时通过标准的 Java 管理扩展(Java Management Extension, JMX)系统来分析 Hibernate 统计数据。将 Hibernate Statistics 对象绑定为一个 MBean；使用动态代理只有几行代码而已。我们已经在 org.jpwh.test.cache.SecondLevel 中包含了一个示例。

正如本节开头所提到过的，Hibernate 会透明地写入和读取缓存的数据。对于某些过程，需要对缓存使用更多的控制，并且可能希望显式绕过缓存。这就是缓存模式可以发挥作用的地方。

20.2.5 设置缓存模式

JPA 用几个缓存模式标准化了共享缓存的控制。例如，以下的 EntityManager#find() 操作不会尝试进行缓存查找和直接访问数据库：

路径: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
Map<String, Object> properties = new HashMap<String, Object>();
properties.put("javax.persistence.cache.retrieveMode",
    CacheRetrieveMode.BYPASS);
Item item = em.find(Item.class, ITEM_ID, properties); ← 访问数据库
```


默认的 `CacheRetrieveMode` 是 `USE`；这里，要使用 `BYPASS` 为一个操作重写它。

更为经常使用的缓存模式是 `CacheStoreMode`。默认情况下，Hibernate 会在调用 `EntityManager#persist()` 时将实体数据放入缓存中。它也会在你从数据库加载实体实例时将数据放入缓存中。不过如果存储或加载大量的实体实例，则可能不希望填满可用的缓存。这对于批处理程序尤为重要，就像我们在本章前面的内容中所介绍的一样。

可以通过在 `EntityManager` 上设置一个 `CacheStoreMode` 来禁用整个工作单元的共享实体缓存中的数据存储：

路径：/examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
em.setProperty("javax.persistence.cache.storeMode", CacheStoreMode.BYPASS);

Item item = new Item(
    // ...
);

em.persist(item);
```

← 未存储在缓存中

我们来看看特殊的缓存模式 `CacheStoreMode.REFRESH`。当使用默认的 `CacheStoreMode.USE` 从数据库中加载一个实体实例时，Hibernate 首先会询问缓存它是否已经具有所加载实体实例的数据。然后，如果缓存已经包含该数据，则 Hibernate 不会将加载的数据放入缓存中。这样就避免了缓存写入，假设缓存读取的成本较低的话。使用 `REFRESH` 模式，Hibernate 就总是会将加载的数据放入缓存中，而不会首先查询缓存。

在一个具有同步分布式缓存的群集中，对所有的缓存节点进行写入通常是一个开销很大的操作。实际上，使用一个分布式缓存，应该将配置属性 `hibernate.cache.use_minimal_puts` 设置为 `true`。这样就会优化二级缓存操作以最小化写入，以更为频繁的读取为代价。不过，如果对于缓存提供程序和架构来说，读取和写入之间没什么区别，那么可能希望使用 `CacheStoreMode.REFRESH` 禁用额外的读取(注意，Hibernate 中有些缓存提供程序可以设置 `use_minimal_puts`：例如，使用 Ehcache 的话，这个设置默认是启用的)。

如你所见，可以在 `find()` 操作上以及为整个 `EntityManager` 设置缓存模式。也可以在 `refresh()` 操作和像提示这样的个体 `Query` 上设置缓存模式，就像 14.5 节中所探讨过的那样。每操作和每查询设置会重写 `EntityManager` 的缓存模式。

该缓存模式仅会影响 Hibernate 内部处理缓存的方式。有时希望程式控制缓存系统：例如，将数据从缓存中移除。

20.2.6 控制共享缓存

用于控制缓存的标准 JPA 接口是 `Cache API`：

路径：/examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
EntityManagerFactory emf = JPA.getEntityManagerFactory();
Cache cache = emf.getCache();

assertTrue(cache.contains(Item.class, ITEM_ID));
cache.evict(Item.class, ITEM_ID);
```

```
cache.evict(Item.class);
cache.evictAll();
```

这是一个简单的 API，并且它仅允许你访问实体数据的缓存区域。需要 `org.hibernate.Cache` API 来访问其他的缓存区域，比如集合和自然标识符缓存区域：

路径：/examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
org.hibernate.Cache hibernateCache =
    cache.unwrap(org.hibernate.Cache.class);

assertFalse(hibernateCache.containsEntity(Item.class, ITEM_ID));
hibernateCache.evictEntityRegions();
hibernateCache.evictCollectionRegions();
hibernateCache.evictNaturalIdRegions();
hibernateCache.evictQueryRegions();
```

你将很少需要这些控制机制。同样，要注意，二级缓存的回收是非事务型的：也就是说，`Hibernate` 不会在回收期间锁定缓存区域。

我们继续介绍 `Hibernate` 缓存系统的最后一部分：查询结果缓存。

20.2.7 查询结果缓存

查询结果缓存默认是禁用的，并且你编写的每一个 JPA、条件或原生 SQL 查询都总是会首先访问数据库。在这一节中，我们将介绍 `Hibernate` 为何要默认禁用查询缓存，然后阐释如何在需要时为特定查询启用它。

以下程序会执行一个 JPQL 查询并且将结果存储在查询结果的特殊缓存区域中：

路径：/examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
String queryString = "select i from Item i where i.name like :n";

Query query = em.createQuery(queryString)           ← ① 启用缓存
    .setParameter("n", "I%")
    .setHint("org.hibernate.cacheable", true);

List<Item> items = query.getResultList();           ← ② 执行查询
assertEquals(items.size(), 3);

QueryStatistics queryStats = stats.getQueryStatistics(queryString); ← ③ 得到详情
assertEquals(queryStats.getCacheHitCount(), 0);
assertEquals(queryStats.getCacheMissCount(), 1);
assertEquals(queryStats.getCachePutCount(), 1);

SecondLevelCacheStatistics itemCacheStats =         ← ④ 将数据存储在实体缓存中
    stats.getSecondLevelCacheStatistics(Item.class.getName());
assertEquals(itemCacheStats.getElementCountInMemory(), 3);
```

① 必须为某个特定查询启用缓存。不使用 `org.hibernate.cacheable` 提示的话，结果就不会被存储到查询结果缓存中。

② `Hibernate` 会执行 SQL 查询并且将结果集检索到内存中。

③ 使用统计 API，就能找到更多的详情。这是第一次执行这个查询，因此会得到缓存未命中而非一次访问。Hibernate 会将查询及其结果放入缓存中。如果再次运行相同的查询，那么结果将来自缓存。

④ 在结果集中检索的实体实例数据会被存储在实体缓存区域中而非查询结果缓存中。

org.hibernate.cacheable 提示是在 Query API 上设置的，因此它也适用于条件和原生 SQL 查询。从内部来说，缓存键是 Hibernate 用来访问数据库的 SQL，它具有在你使用参数标记的字符串中呈现的参数。

查询结果缓存不会包含 SQL 查询的整个结果集。在最后一个示例中，SQL 结果集包含了 ITEM 表的行。Hibernate 会忽略这个结果集中的大多数信息；只会将每条 ITEM 记录的 ID 值存储到查询结果缓存中。每个 Item 的属性值都会被存储到实体缓存区域中。

现在，当将相同的参数值用于相同的查询参数再次执行该相同查询时，Hibernate 会首先访问查询结果缓存。它会从用于查询结果的缓存区域中检索 ITEM 记录的标识符值。然后，Hibernate 会根据标识符从实体缓存区域中查找和装配每个 Item 实体实例。如果查询实体并且决定启用缓存，那么要确保你同时为这些实体启用常规数据缓存。如果不是这样，那么在启用查询结果缓存之后，你可能最终会得到更多的数据库访问。

如果缓存不会返回实体实例而是只返回标量或可嵌入值的查询结果(例如，select i.name from Item i 或 select u.homeAddress from User)，那么这些值会被直接保留在查询结果缓存区域中。

查询结果缓存使用了两个物理缓存区域：

路径：/model/src/main/resources/cache/ehcache.xml

```
<cache name="org.hibernate.cache.internal.StandardQueryCache"
      maxElementsInMemory="500"
      eternal="false"
      timeToIdleSeconds="600"
      timeToLiveSeconds="3600"/>
```

```
<cache name="org.hibernate.cache.spi.UpdateTimestampsCache"
      maxElementsInMemory="50"
      eternal="true"/>
```

第一个缓存区域是存储查询结果的地方。随着时间的推移，你应该让缓存提供程序将最近使用过的结果集过期，这样缓存就会将可用空间用于最近执行的查询。

第二个区域，org.hibernate.cache.spi.UpdateTimestampsCache，是特殊的：Hibernate 会使用这个区域来判定缓存的查询结果集是否过时。当启用了缓存重新执行一个查询时，Hibernate 就会为对查询表所做的最近插入、更新或删除的时间戳查找时间戳缓存区域。如果找到的时间戳晚于所缓存的查询结果的时间戳，那么 Hibernate 就会丢弃该缓存结果并且发出一条新的数据库查询。如果查询中可能涉及的任何表包含更新的数据，那么这实际上将确保 Hibernate 不会使用缓存的查询结果；因此，缓存的结果可能是过时的。应该禁用更新时间戳缓存的过期设置，以便缓存提供程序永远不会从这个缓存中移除一个元素。这个缓存区域中的元素最大数量取决于你的映射模型中的表数量。

大多数查询不会受益于结果缓存。这可能会让你感到吃惊。毕竟，避免数据库访问听起来总是一件好事情。与根据标识符的实体检索或者集合初始化对比，这并不总是适用于任意查询的合理理由有两个。

首先，必须想想你打算使用相同参数重复执行相同查询的频率。的确，应用程序可以重复执行正好具有绑定到形参的相同实参的一些查询以及相同的自动生成的 SQL 语句。我们认为这是一个罕见的用例，但当确定你正在重复执行一个查询时，它对于结果集缓存就会变成一个合适的候选。

其次，对于执行许多查询和少数插入、删除或更新的应用程序来说，缓存查询结果可以提升性能和可扩展性。另一方面，如果应用程序执行许多写入，那么 Hibernate 实际上就不会使用查询结果缓存。Hibernate 会在缓存的查询结果中出现任何插入、更新或删除一个表的任意行时将缓存的查询结果集过期。这意味着缓存的结果可能有一个短的生命周期，并且即便你重复执行一个查询，Hibernate 也不会使用缓存的结果，因为由查询所引用的表中的行存在并发修改。

对于许多查询来说，查询结果缓存的好处在于不存在或者至少没有你所预期的影响。但如果查询限制位于一个唯一自然标识符上，比如 `select u from User u where u.username = ?`，那么你应该考虑本章前面内容中介绍过的自然标识符缓存和查找。

20.3 本章小结

- 介绍了扩展应用程序和处理许多并发用户以及较大数据集所需的选项。
- 使用大批量 UPDATE 和 DELETE 操作，你就可以直接在数据库中修改数据并且仍旧能从 JPQL 和条件 API 中获益，而无须回退到 SQL。
- 介绍了可以在应用层中处理大量记录的批处理操作。
- 详尽探讨了 Hibernate 缓存系统：如何才能选择性地启用和优化实体、集合以及查询结果数据的共享缓存。
- 配置了 Ehcache 作为缓存提供程序，并且学习了如何使用 Hibernate 统计 API 探究背后的情况。

Hibernate 实战 (第2版)

Java Persistence with Hibernate Second Edition

持久化——数据在程序实例之外留存的功能——是现代应用程序的核心。Hibernate是最流行的Java持久化工具，提供了自动且透明的对象/关系映射，使得在Java应用程序中使用SQL数据库变得轻而易举。

《Hibernate实战(第2版)》通过开发一个将数百个单独示例联系起来的应用程序来探究Hibernate。你将直接深入到Hibernate的富编程模型之中，贯穿映射、查询、抓取策略、事务、会话、缓存以及更多其他内容。书中图文并茂地介绍了数据库设计和优化技术的最佳实践。在本书中，作者详尽介绍了具有Java持久化2.1标准的Hibernate 5(JSR 338)。所有的示例都已经被更新，以便用于最新的Hibernate和Java EE规范版本。

主要内容

- ◆ 对象/关系映射概念
- ◆ 有效的数据库应用程序设计
- ◆ 全面的Hibernate与Java持久化介绍
- ◆ Java持久化与EJB、CDI、JSF和JAX-RS的集成
- ◆ 无与伦比的广度和深度

本书假设读者具有Java的使用经验。

作者简介

Christian Bauer是Hibernate开发团队的成员，并且是一位培训师和顾问。Gavin King是Hibernate之父，也是Java持久化专家组(JSR 220)的成员。Gary Gregory是应用程序服务器和既有系统集成的首席软件工程师。

源代码下载



清华大学出版社数字出版网站

WQBook 书文局泉
www.wqbook.com

MANNING

“关于Hibernate持久化最为全面的一本书……既可充当教程也可用作参考书。”

—Sergio Fernandez Gonzalez
Accenture Software

“引领你应对Hibernate复杂性的必备指南。”

—José Diaz, OptumHealth

“一本经典及必备书籍的优秀的更新版本。”

—Jerry Goodnough
Cognitive Medical Systems

“每一位Hibernate用户都必须拥有的参考书籍。”

—Stephan Heffner
SPIEGEL-Verlag Rudolf Augstein
GmbH & Co. KG

ISBN 978-7-302-44808-2



9 787302 448082

定价：88.00元